Oracle® Application Server

Developer’s Guide: LiveHTML and Perl Applications

Release 4.0.8.1

September 1999

Part No. A66960-02

Oracle Application Server Release 4.0.8.1 Developer's Guide: LiveHTML and Perl Applications

Part No. A66960-02

Copyright © 1996, 1999, Oracle Corporation. All rights reserved.

Primary Author: Janice Nygard

Contributing Authors: Kai Li

Contributors: Sanjay Patil

The programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be the licensee's responsibility to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use of such applications if the programs are used for such purposes, and Oracle Corporation disclaims liability for any damages caused by such use of the programs.

The programs (which include both the software and documentation) contain proprietary information of Oracle Corporation; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent, and other intellectual and industrial property laws. Reverse engineering, disassembly, or decompilation of the programs is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. Oracle Corporation does not warrant that this document is error free. Except as may be expressly permitted in your license agreement for these programs, no part of these programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Oracle Corporation.

If the programs are delivered to the U.S. Government or anyone licensing or using the programs on behalf of the U.S. Government, the following notice is applicable:

Restricted Rights Notice Programs delivered subject to the DOD FAR Supplement are "commercial computer software" and use, duplication, and disclosure of the programs, including documentation, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement. Otherwise, programs delivered subject to the Federal Acquisition Regulations are "restricted computer software" and use, duplication, and disclosure of the programs shall be subject to the restrictions in FAR 52.227-19, Commercial Computer Software - Restricted Rights (June, 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

Oracle is a registered trademark, and the Oracle logo, NLS*WorkBench, Pro*COBOL, Pro*FORTRAN, Pro*Pascal, SQL*Loader, SQL*Module, SQL*Net, SQL*Plus, Oracle7, Oracle Server, Oracle Server Manager, Oracle Call Interface, Oracle7 Enterprise Backup Utility, Oracle TRACE, Oracle WebServer, Oracle Web Application Server, Oracle Application Server, Oracle Network Manager, Secure Network Services, Oracle Parallel Server, Advanced Replication Option, Oracle Data Query, Cooperative Server Technology, Oracle Toolkit, Oracle MultiProtocol Interchange, Oracle Names, Oracle Book, Pro*C, and PL/SQL are trademarks or registered trademarks of Oracle Corporation. All other company or product names mentioned are used for identification purposes only and may be trademarks of their respective owners.

Contents

4 Writing Scripts

13 Writing Perl Scripts

14 Upgrading your Perl Interpreter

15 Troubleshooting

Preface

Audience

This book is written for users who develop web applications using the LiveHTML and Perl cartridges of Oracle Application Server.

The Oracle Application Server Documentation Set

This table lists the Oracle Application Server documentation set.

Title of Book	Part No.
Oracle Application Server 4.0.8 Documentation Set	A66971-03
Oracle Application Server Overview and Glossary	A60115-03
Oracle Application Server Installation Guide for Sun SPARC Solaris 2.x	A58755-03
Oracle Application Server Installation Guide for Windows NT	A58756-03
Oracle Application Server Administration Guide	A60172-03
Oracle Application Server Security Guide	A60116-03
Oracle Application Server Performance and Tuning Guide	A60120-03
Oracle Application Server Developer’s Guide: PL/SQL and ODBC Applications	A66958-02
Oracle Application Server Developer’s Guide: JServlet Applications	A73043-01
Oracle Application Server Developer’s Guide: LiveHTML and Perl Applications	A66960-02
Oracle Application Server Developer’s Guide: EJB, ECO/Java and CORBA Applications	A69966-01
Oracle Application Server Developer’s Guide: C++ CORBA Applications	A70039-01
Oracle Application Server PL/SQL Web Toolkit Reference	A60123-03
Oracle Application Server PL/SQL Web Toolkit Quick Reference	A60119-03

Title of Book	Part No.
Oracle Application Server JServlet Toolkit Reference	A73045-01
Oracle Application Server JServlet Toolkit Quick Reference	A73044-01
Oracle Application Server Cartridge Management Framework	A58703-03
Oracle Application Server 4.0.8.1 Release Notes	A66106-04

Conventions

This table lists the typographical conventions used in this manual.

Convention	Example	Explanation
bold	**oas.h** **owsctl** **wrbcfg** **www.oracle.com**	Identifies file names, utilities, processes, and URLs
italics	*file1*	Identifies a variable in text; replace this place holder with a specific value or string.
angle brackets	`<filename>`	Identifies a variable in code; replace this place holder with a specific value or string.
courier	`owsctl start wrb`	Text to be entered exactly as it appears. Also used for functions.
square brackets	`[-c string]`	Identifies an optional item.
	`[on|off]`	Identifies a choice of optional items, each separated by a vertical bar (|), any one option can be specified.
braces	`{yes|no}`	Identifies a choice of mandatory items, each separated by a vertical bar (|).
ellipses	n,...	Indicates that the preceding item can be repeated any number of times.

The term “Oracle Server” refers to the database server product from Oracle Corporation.

The term **“oracle**” refers to an executable or account by that name.

The term “*oracle*” refers to the owner of the Oracle software.

Technical Support Information

Oracle Global Support can be reached at the following numbers:

- In the USA: **Telephone: 1.650.506.1500**
- In Europe: **Telephone: +44 1344 860160**
- In Asia-Pacific: **Telephone: +61. 3 9246 0400**

Please prepare the following information before you call, using this page as a checklist:

❑ your CSI number (if applicable) or full contact details, including any special project information

❑ the complete release numbers of the Oracle Application Server and associated products

❑ the operating system name and version number

❑ details of error codes and numbers and descriptions. Please write these down as they occur. They are critical in helping WWCS to quickly resolve your problem.

❑ a full description of the issue, including:

  - **What -** What happened? For example, the command used and its result.
  - **When -**When did it happen? For example, during peak system load, or after a certain command, or after an operating system upgrade.
  - **Where -**Where did it happen? For example, on a particular system or within a certain procedure or table.
  - **Extent -** What is the extent of the problem? For example, production system unavailable, or moderate impact but increasing with time, or minimal impact and stable.

❑ Keep copies of any trace files, core dumps, and redo log files recorded at or near the time of the incident. WWCS may need these to further investigate your problem. For a list of trace and log files, see “Configuration and Log Files” in the *Administration Guide.*

For installation-related problems, please have the following additional information available:

❑ listings of the contents of $ORACLE_HOME (Unix) or %ORACLE_HOME% (NT) and any staging area, if used.

- ❑ installation logs (**install.log**, **sql.log**, **make.log**, and **os.log**) typically stored in the **$ORACLE_HOME/orainst** (Unix) or **%ORACLE_HOME%\orainst** (NT) directory.

Documentation Sales and Client Relations

In the United States:

- To order hardcopy documentation, call Documentation Sales: **1.800.252.0303.**
- For shipping inquiries, product exchanges, or returns, call Client Relations: **1.650.506.1500.**

In the United Kingdom:

- To order hardcopy documentation, call Oracle Direct Response: **+44 990 332200.**
- For shipping inquiries and upgrade requests, call Customer Relations: **+44 990 622300.**

Reader's Comment Form

Oracle Application Server 4.0 Developer's Guide: LiveHTML and Perl Applications
Part No. A66960-02

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have suggestions for improvement, please indicate the topic, chapter, and page number below:

Please send your comments to:

Oracle Application Server Documentation Manager
Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065

If you would like a reply, please provide your name, address, and telephone number below:

Thank you for helping us improve our documentation.

Part I

LiveHTML Cartridge

1

LiveHTML Cartridge Overview

The LiveHTML cartridge simplifies the task of generating dynamic HTML pages. When you want to generate a HTML page dynamically, you usually have to write a script or program to generate the entire page including the static portions. This requires more time to write the scripts and programs and to generate each dynamic page.

LiveHTML provides an alternative method of generating dynamic HTML pages. It saves you from generating the entire HTML page each time it is requested by allowing you to embed server-side includes (SSI commands) and scripts in an otherwise static HTML page.

The LiveHTML cartridge is Oracle Application Server's implementation and extension of SSI and server-side scripts. The cartridge allows you to embed SSI commands and scripts directly within HTML pages.

The LiveHTML cartridge features:

- Server-Side Includes (SSI)
- Embedded Scripts
- Web Application Objects
- IDL-to-Perl Compiler
- Process Flow

Server-Side Includes (SSI)

Using SSI, you can:

- include other files in the current file
- get the values of CGI and SSI environment variables

- get the current date and time
- get the size of a file
- get the last modification date of a file
- run scripts
- send requests to other cartridges (this is an extension to the standard SSI). Using Oracle Application Server's Inter-cartridge Exchange service (ICX), you can send a request to other cartridges, such as the PL/SQL cartridge or the Java cartridge, and include the data returned from the cartridges in the page.

Embedded Scripts

The LiveHTML cartridge is able to process scripts embedded within HTML pages. This scripting feature enables you to combine static HTML with the full functionality of scripting languages. Currently, the cartridge runtime supports Perl as the scripting language.

In your scripts, you can use all of Perl's features. For example, you can call Perl's built-in functions, define and invoke your own functions, include Perl modules in your scripts (with the `use` statement), and use Perl's special variables. See Chapter 4, "Writing Scripts" for details and script examples.

Note: The LiveHTML cartridge uses the same interpreter as the Perl cartridge. The implementation of this interpreter has functional variations from standard Perl interpreters. Refer to the section "Variations from Perl Standard Version" in Chapter 10.

Scripts in your page can also access Web Application Objects. These objects help you to manage and design your HTML pages as an application.

The scripting feature is compatible with Microsoft Active Server Pages. If you have Active Server Pages written in Perl, you can use them with the LiveHTML cartridge.

Web Application Objects

Scripts in your LiveHTML pages can access Web Application Objects, a framework designed to provide robust support for building transactional web-based applications. See Chapter 5, "Developing with Web Application Objects" for details.

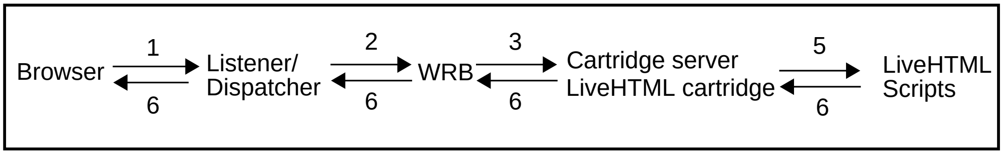
IDL-to-Perl Compiler

An IDL-to-Perl compiler is provided with the LiveHTML cartridge. This compiler, **perlidlc**, allows you to generate Perl bindings for CORBA objects so that they can be used in Perl scripts in LiveHTML pages. Refer to Chapter 7, “Accessing CORBA Objects from Perl Scripts” for more details.

Process Flow

The following figure illustrates the flow of a request for a LiveHTML application.

Figure 1–1 Process flow for a request to a LiveHTML application



1. The Listener component of Oracle Application Server receives a request for a LiveHTML cartridge from a client.
2. The Dispatcher sees that the request is for a cartridge and forwards it to the WRB.
3. The WRB examines the URL and sends the request to a LiveHTML cartridge since the virtual path specified in the URL is mapped to a LiveHTML cartridge.
4. The LiveHTML cartridge running in a cartridge server process receives the request, examines the URL, and finds the name of the LiveHTML file to parse.
5. The LiveHTML cartridge loads the file and invokes the scripting engine to interpret the scripts in the file.
6. The scripting engine generates a response, including both the HTTP response header and response body, and returns it to the LiveHTML cartridge. The Live-

HTML cartridge receives the response and returns it to the WRB. The WRB forwards the response to the client browser that invoked the request.

Note: Currently, the NT environment has a 512 byte limitation on the expanded length of some environment variables (CLASSPATH, JAVA_HOME, etc.). Some Oracle Application Server cartridges will try to expand environment variables. Therefore, make sure that your environment variables are not longer than 250-300 characters long.

2

Adding and Invoking Applications

This chapter describes how to add LiveHTML applications to Oracle Application Server and invoke them from browsers.

Contents

Adding LiveHTML Applications

To add Oracle Application Server applications to the application server, you perform these general steps:

- Add the application
- Add cartridge(s) to the application

To add applications and cartridges, you need to be able to log in as the “admin” user in the Oracle Application Server Manager.

To add applications and cartridges:

1. Start up your browser and display the top-level administration page for Oracle Application Server.
2. Click the ⊞ next to a site name to display the components on the site. You should see “Oracle Application Server”, “HTTP Listeners”, and “Applications”.
3. Click “Applications” to display the applications in the right frame. Do not click the ⊞ next to Applications because you will see a list of applications for the site in the left frame, instead of Applications in the right frame.

4. On the applications page in the right frame, click ➕. This pops up the Add Application dialog.
5. In the Add Application dialog:
   - Application Type: select LiveHTML.
   - Configure Mode: select Manual, which enables you to enter configuration data using dialog boxes. The other option, From File, assumes that you have already entered the configuration data for the application in a file.
   - Click Apply.

   This displays the Add Application dialog.
6. In the Add Application dialog:
   - Application Name: enter the name that the server uses to identify your application.
   - Display Name: enter the name that is used in the administration forms.
   - Application Version: enter a version number for your application.
   - Click Apply.

     When you click Apply, you get a Success dialog, which contains a button that enables you to add LiveHTML cartridges to the application.
7. In the Success dialog box, click the Add Cartridges to Application button. This displays the Add A Cartridge dialog.
8. In the Add A Cartridge dialog:
   - Cartridge Name: enter the name that the server uses to identify your LiveHTML cartridge in your application.
   - Display Name: enter the name that is used in the administration forms.
   - Virtual Path: enter a path for the LiveHTML cartridge such that users can specify this path in URLs to invoke the LiveHTML cartridge. This path is mapped to the physical path that you specify below. See the section Virtual Paths Form below.
   - Physical Path: enter the physical directory path that leads to files for your LiveHTML cartridge, including files for your LiveHTML application. The virtual path specified above maps to this physical path.

Note: For security reasons, you cannot specify a physical path ending with "`..`". But you can use "`..`" in the physical path setting to indicate an upper directory level. For example, "`/routines/../libraries/`".

---

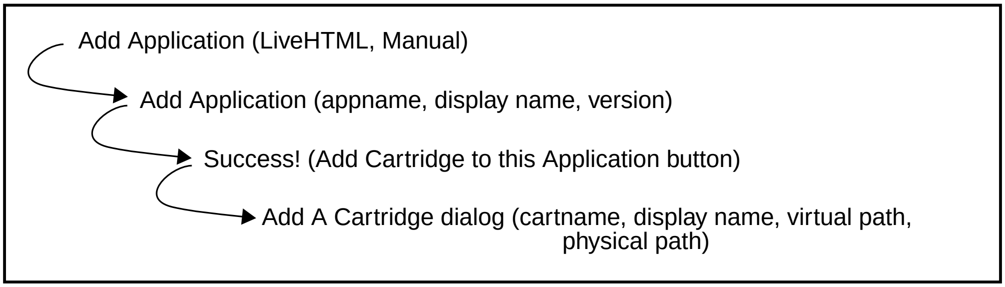
9. Click Apply.
10. Stop and restart the listeners and other components of the application server.

    See "Stopping and Starting the Application Server" in the "Application Configuration" chapter for details.

Note: If your application does not appear in the navigational tree, shift-click or control-click the browser's Reload button.

The following figure summarizes the dialog boxes that you completed. The fields in the dialog boxes are listed in parentheses.

Figure 2–1 Dialogs to create a LiveHTML application and cartridge


Add Application (LiveHTML, Manual)
Add Application (appname, display name, version)
Success! (Add Cartridge to this Application button)
Add A Cartridge dialog (cartname, display name, virtual path, physical path)

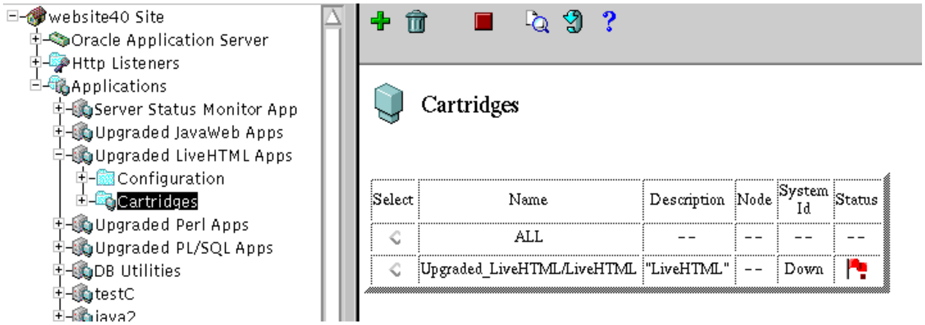
Adding Cartridges to an Existing Application

A LiveHTML application can have one or more cartridges. You need more than one cartridge in a LiveHTML application if you need to configure cartridge parameters differently for each cartridge. For example, you might enable ICX in some cartridges but not others.

To add a cartridge to a LiveHTML application:

1. Select "Cartridges" under the LiveHTML application to which you want to add cartridges in the navigational tree.

Figure 2–2 Adding LiveHTML cartridges to an existing application



Cartridges

Select	Name	Description	Node	System Id	Status
○	ALL	--	--	--	--
○	Upgraded_LiveHTML/LiveHTML	"LiveHTML"	--	Down	🚩

2. Click ✚ to display the Add Cartridge dialog.
3. In the Add Cartridge dialog:
   - Configure Mode: select Manually.
   - Click Apply, which displays the Add A Cartridge dialog.
4. In the Add A Cartridge dialog:
   - Cartridge Name: enter the name that the server uses to identify your LiveHTML cartridge in your application.
   - Display Name: enter the name that is used in the administration forms.
   - Virtual Path: enter a path for the LiveHTML cartridge such that users can specify this path in URLs to invoke the LiveHTML cartridge. This path is mapped to the physical path that you specify below. See the section Virtual Paths Form below.
   - Physical Path: enter the physical directory path that leads to files for your LiveHTML cartridge, including files for your LiveHTML application. The virtual path specified above maps to this physical path.

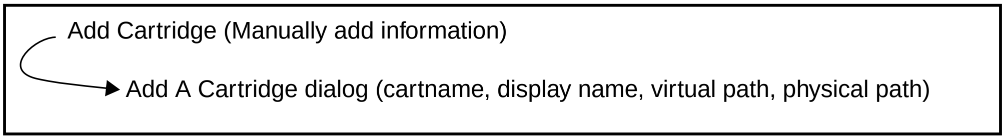
   > **Note:** For security reasons, you cannot specify a physical path ending with "`..`". But you can use "`..`" in the physical path setting to indicate an upper directory level. For example, "`/routines/../libraries/`".

   - Click Apply.

Note: If your new cartridge does not appear in the navigational tree, shift-click or control-click the browser's Reload button.

The following figure summarizes the dialog boxes that you completed. The fields in the dialog boxes are listed in parentheses.

Figure 2–3 Dialogs to add LiveHTML cartridges



Add Cartridge (Manually add information)

Add A Cartridge dialog (cartname, display name, virtual path, physical path)

Configuring LiveHTML Applications

The configuration forms are divided into two sections: application configuration and cartridge configuration. Forms in the application configuration section contain parameters that apply for the entire application, while forms in the cartridge configuration section contain parameters that apply to a particular cartridge.

Application Configuration

Application configuration parameters are described in the "Application Configuration" chapter, because they are the same for all types of applications.

Cartridge Configuration

For LiveHTML cartridges, the cartridge configuration section contains two forms: the Virtual Paths form and the LiveHTML Parameters form.

Virtual Paths Form

The Virtual Paths form enables you to specify a virtual path for a LiveHTML cartridge. Users can then specify this virtual path in URLs to invoke the LiveHTML cartridge. The virtual path is available from all listeners listed in the Web Configuration page for the application.

For example, if you specify a virtual path of **/myApp**, users can invoke your LiveHTML applications by typing **/myApp/***file* at the URL, where *file* is a filename that can be found in the physical paths associated with the **/myApp** virtual path.

Note: For security reasons, you cannot specify a physical path ending with "`..`". But you can use "`..`" in the physical path setting to indicate an upper directory level. For example, "`/routines/../libraries/`".

LiveHTML Parameters Form

The LiveHTML Parameters form (Figure 2–4) enables you to configure parameters specific to LiveHTML cartridges. It has the following parameters:

Table 2–1 LiveHTML-specific configuration

Option	Description
Enable LiveHTML	Whether or not the LiveHTML cartridge is enabled. If not enabled, SSI commands, scripting commands, and Web Application Objects are not interpreted. If you want to enable only some features of the cartridge, you can enable the LiveHTML cartridge, but disable the features that you do not want. **Default**: Enabled
Parse LiveHTML Extensions Only	Whether or not the cartridge should parse files with the extensions specified in the “LiveHTML Extensions” field. If enabled, the cartridge will parse files with extensions listed in the “LiveHTML Extensions” field only. If not enabled, the cartridge will parse all files regardless of extension. **Default**: Enabled
LiveHTML Extensions	The list of file extensions handled by the cartridge. This field is used only if you have enabled the “Parse Live-HTML Extensions Only” field. You can configure the cartridge to process all HTML files (that is, set the extension list to include “html”). However, unless all your HTML files actually use SSI, this degrades performance. **Default**: `html shtml lhtml`

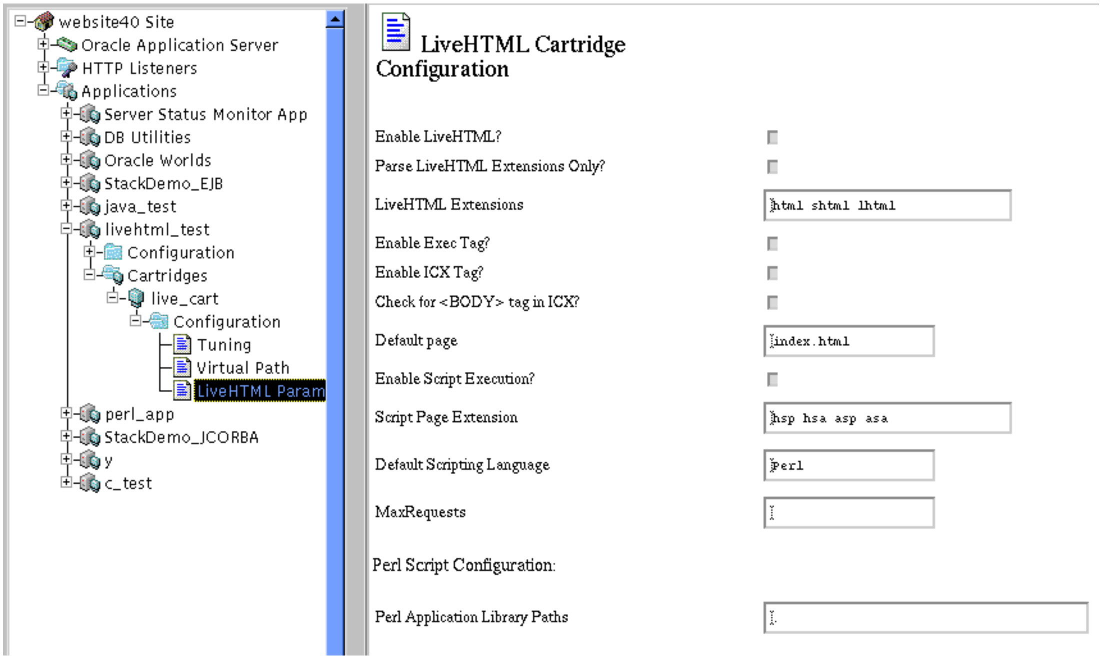
Table 2–1 LiveHTML-specific configuration

Option	Description
Enable Exec Tag	Whether or not the exec SSI command is interpreted by the cartridge. **Default**: Enabled
Enable ICX Tag	Whether or not the request command is interpreted by the cartridge. **Default**: Enabled
Check for <BODY> tag in ICX	Whether or not the cartridge checks for the <BODY> tag inside the response to an ICX request. (ICX requests are sent using the request command.) If enabled, only data in the <BODY> section of the ICX response is included in the page that sent the request command. If no <BODY> section is found in the ICX response, the cartridge raises an error. If not enabled, the entire ICX response is included in the page. **Default**: Enabled
Default page	The page that is returned to the client if the URL does not specify a file. **Default**: `index.html`
Enable Script Execution	Whether or not embedded scripts in the files are interpreted by the cartridge. **Default**: Enabled
Script Page Extension	The list of file extensions that the cartridge checks for embedded scripts. **Default**: `hsp hsa asp asa`
Default Scripting Language	The default scripting language. Currently, Perl is the only language supported. You can specify a different language for the scope of a page or script block. See "Specifying Scripting Languages" in Chapter 4. **Default**: `Perl`

Table 2–1 LiveHTML-specific configuration

Option	Description
Max Requests	The number of requests that a cartridge server handles before it terminates. This field can be useful while you are developing LiveHTML applications. If your page calls a Perl library, the Perl interpreter caches the Perl library and uses the cached version for subsequent requests. If you modify the library, you want the interpreter to load the new version. To do this, you have to terminate the cartridge server process so that a new cartridge server process (with a new Perl interpreter) would handle the request. A quick way of doing this is to set the Max Requests value to 1. **Default**: There is no default, which means that the cartridge server can handle an unlimited number of requests.
Perl Application Library Paths	The directories that the Perl interpreter searches in for Perl libraries. If you add paths to this option, you should use full pathnames. If specifying multiple directories, use "`:`" to delimit each directory. **Default**: `.` (the current working directory of the cartridge server process)

Figure 2–4 LiveHTML cartridge configuration form



LiveHTML Cartridge Configuration

Enable LiveHTML?	□
Parse LiveHTML Extensions Only?	□
LiveHTML Extensions	html shtml lhtml
Enable Exec Tag?	□
Enable ICX Tag?	□
Check for <BODY> tag in ICX?	□
Default page	index.html
Enable Script Execution?	□
Script Page Extension	hsp hsa asp asa
Default Scripting Language	Perl
MaxRequests	

Perl Script Configuration:

Perl Application Library Paths	

Security

Enabling users to execute scripts on the server can create security problems and other risks. You can configure the LiveHTML cartridge in the following ways to minimize security risks:

- You can configure the cartridge not to run any SSI commands that involve executing scripts. The cartridge can only get the values of environment variables or it can include other files. You do this by disabling the exec tag.
- You can also configure it not to run any requests that invoke another cartridge. You do this by disabling the ICX tag.
- To be more extreme, you can disable the cartridge entirely, in which case no SSI commands or scripts are interpreted. You do this by disabling the LiveHTML option.
- You can protect the virtual paths for LiveHTML cartridges using authentication and restriction schemes. See the *Security Guide* for details.

3

Using Server-Side Includes

Contents

SSI Commands

Server-Side Includes (SSI) commands are formatted as HTML comments and have the following format:

```
<!--#command [param1="value1" param2="value2" ...] -->
```

where:

- *command* is the name of the SSI command
- *param1* and *param2* are names of parameters to pass to the command
- *value1* and *value2* are values for the parameters

Parameters depend on the command; some commands do not take parameters.

In your LiveHTML files, you can have only one SSI command per line. For example:

```
<p><!--#include file="notes.htm"-->  Server name: <!--#echo var="SERVER_NAME" -->
```

must be broken into two separate lines:

```
<p><!--#include file="notes.htm"-->
Server name: <!--#echo var="SERVER_NAME" -->
```

The output is the same.

Errors

If an error occurs while processing an SSI command (for example, the specified command was not found or there was a parsing error), the user sees the following message: “Server Side Processing Error”.

You can change this error message using the `config errmsg` command. For example, the following command sets the error message to “If you see this message, an error occurred.”

```
<!--#config errmsg="If you see this message, an error occurred.">
```

The custom message is used for all errors.

Special Characters

The $ and the single quote characters have special meaning to the LiveHTML cartridge. If you need to specify these characters literally within an SSI tag, precede them with a backslash (\).

Command Summary

The following table lists the SSI commands that can be processed by the LiveHTML cartridge:

Table 3–1 SSI commands

Command	Description
config	Sets parameters for how the included files or scripts are to be parsed. It is normally the first LiveHTML command in a file.
include	Specifies that a file is to be included in the generated HTML page at this point.

Table 3–1 SSI commands

Command	Description
echo	Gives the value of an environment variable.
fsize	Gives the size of the file.
flastmod	Gives the last modification date of the file.
exec	Executes a script.
request	Sends a request to another cartridge using inter-cartridge exchange (ICX).

config

Defines formatting information for other SSI commands in the file. The `config` command is usually the first SSI command in a file; you can have more than one `config` command in a file.

Parameter	Description
errmsg	The error message that is sent to the client if an error occurs while parsing the document.
timefmt	The format to use when displaying dates. The conventions follow the `strftime` library call. (Refer to your system's `strftime` man pages for more information.) Ordinary characters in the format are copied to the document without conversion, so you can insert "on" or "at" or other useful strings.
sizefmt	The format to use when displaying file size. Possible values are: ▪ `bytes` - the file size is given in bytes ▪ `abbrev` - the file size is given in Kb or Mb
cmdecho	Whether non-CGI scripts subsequently executed have their output incorporated into this HTML page. The possible values are `ON` and `OFF`. `ON` specifies that the output is included. The default is `OFF`.
cmdprefix	The string to prepend to each line of the script output.
cmdpostfix	The string to append to each line of the script output.

Example:

```
<!--#config errmsg="A parse error occurred" sizefmt="bytes" cmdecho="ON"-->
```

include

Includes a file in the generated HTML page. The file can be another LiveHTML file, a regular HTML file, or an ASCII file. Oracle Application Server determines the type of the included file by its extension.

Parameter	Description
virtual	The virtual path to the file. The directory mappings for virtual paths are set by the Oracle Application Server administrator using Oracle Application Server Manager.
file	The pathname relative to the current directory. References to parent directories or uses of absolute pathnames are not allowed.

If the included file is a complete HTML document, the LiveHTML cartridge reads only the data in the <BODY> section of the document. Data in the <HEAD> section of the document is not included.

Example:

You can include a set of common links in your files by including a file that specifies the links. If you insert this command in your HTML file:

```
<!--#include file="links.html"-->
```

and the **links.html** file contains:

```
<p><a href="home.html">Home</a> |
<a href="index.html">Index</a> |
<a href="search.html">Search</a>
```

this would result in the following links being inserted into your Web page:

```
Home | Index | Search
```

echo

Gives the value of a standard CGI or SSI environment variable.

Parameter	Description
var	The name of the environment variable.

For a list of CGI environment variables, see **http://hoohoo.ncsa.uiuc.edu/cgi-1.1/**.

The following table lists the SSI environment variables:

Table 3–2 SSI environment variables

SSI environment variable	Description
DOCUMENT_NAME	The current filename.
DOCUMENT_URI	The virtual path to this file.
QUERY_STRING_UNESCAPED	If the client sent a query string, this is an unescaped version of it, with all shell-special characters escaped with \.
DATE_LOCAL	The current date and local time zone, given in the format specified in the most recent config timefmt command.
DATE_GMT	The current date and time zone in Greenwich Mean Time, given in the format specified in the most recent config timefmt command.
LAST_MODIFIED	The last modification date of the file, given in the format specified in the most recent config timefmt command.

Example:

The command:

```
Current date/time: <!--#echo var="DATE_LOCAL"-->
```

generates something like:

```
Current date/time: Thursday, April 17, 1997 03:15 PM
```

fsize

Gives the size of the file in the format specified in the most recent “config sizefmt” command.

This command takes the same parameters as include.

Example:

The command

```
<!--#fsize file="logo.jpg"-->
```

generates the size of the logo.jpg file.

A common use of this command is to provide the user with the sizes of graphic files to be downloaded. This is a tremendous time saver if you add and change downloadable images frequently because you never have to look up the file sizes and enter them manually.

flastmod

Gives the last modification date of the file in the format specified in the most recent “`config timefmt`” command.

This command takes the same parameters as `include`.

Example:

The command

```
<!--#flastmod file="releases.html"-->
```

generates the date when the releases.html file was last modified.

exec

Executes a script. The parameter specifies whether or not the script is CGI.

Note that before you can use the `exec` command, you need to enable it in the LiveHTML cartridge configuration. In the Cartridge Parameters page of the LiveHTML cartridge, set the `EnableExecTag` to `TRUE`.

Parameter	Description
`cmd`	Specifies a non-CGI script. Execution is passed to the operating system, and the given string is parsed as though it were entered at a command-line interface. The full path of the script must be given. You can reference the SSI environment variables. **Note:** For the output of the script to be included in the HTML page, you have to set the following command in the page: `<!--#config cmdecho="ON"-->`
`cgi`	Specifies a CGI script. The value is the virtual path of the CGI script. URL locations are automatically converted into HTML anchors.

Example:

```
<!--#config cmdecho="ON" -->
...
<!--#exec cmd="/bin/who" -->
```

request

Makes an ICX (inter-cartridge exchange) request. The ICX feature of Oracle Application Server allows cartridges to communicate with each other by making HTTP requests. For example, you can use this command to send a request to a PL/SQL cartridge to run a stored procedure in the database, and embed the results of the stored procedure in the LiveHTML document.

The `request` command is an Oracle Application Server extension to the SSI command set.

> **Note:** Before using the `request` command, you need to enable it in the LiveHTML cartridge configuration. In the Cartridge Parameters page of the LiveHTML cartridge, set the `EnableICXTag` to `TRUE`.

Parameter	Description
`url`	The URL to which to send the request. The syntax of the URL is: http://*user*:*password*@*host*:*port*/*url-path*?*QS* where: *url-path* extends the semantics of the common URL because the LiveHTML cartridge supports variable substitution. *QS* is a query string in the form of name-value pairs. The syntax is the same as the query string in the GET method (each name-value pair is separated by the & character, spaces are replaced with the + character). You can let the LiveHTML cartridge encode the query string for you if you enclose it in single quotes.

Using Values from the Query String

You can invoke a LiveHTML page with name-value pairs in the query string and reference the values in `request` commands on the page. You can use this method to generate a dynamic URL in the `request` command.

For example, if your LiveHTML page is **showUserProp.shtml** and it needs a value for user and a value for property, you can invoke the page with:

```
http://domain/showUserProp.shtml?user=chris&property=job+title
```

This URL can be generated automatically by an HTML form that allows the user to enter the username and the property in form fields.

A `request` command in the **showUserProp.shtml** page can reference the user and property variables by preceding the variable name with the $ character. In this example, the variables are $user and $property.

A `request` command in the **showUserProp.shtml** page could look like:

```
<!--#request URL="/$user/work?SQLString=
                 'select $property from employee'&name=$user" -->
```

The LiveHTML cartridge expands the above request to:

```
/chris/work?SQLString=select+job%20title+from+employee&name=chris
```

The LiveHTML cartridge expects content within single quotes to be a non-encoded URL since it performs the encoding. You must correctly encode the rest of the HTTP URL.

SSI Examples

- Displaying Date and Time
- Getting Information About the Current File
- Getting Information About Other Files
- Displaying Browser Information
- Providing Host and Server Information
- Accessing a Database

Displaying Date and Time

The following config command defines a date and time format, which is used by the echo command.

```
<!--#config timefmt="%A, %B %d, %Y, at %I:%M %p"-->
<p>GMT date/time is
<!--#echo var="DATE_GMT"-->
<p>LOCAL date/time is
<!--#echo var="DATE_LOCAL"-->
<p>Updated on
<!--#echo var="LAST_MODIFIED"-->
```

This produces the following:

```
GMT date/time is Friday, August 23, 1996, at 03:14 AM
LOCAL date/time is Thursday, August 22, 1996, at 08:14 PM
Updated on Tuesday, August 13, 1996, at 03:42 AM
```

Getting Information About the Current File

Given the following lines:

```
This document is <!--#echo var="PATH_TRANSLATED"-->
Its virtual path is <!--#echo var="DOCUMENT_URI"-->
```

On UNIX, you would see:

```
This document is /oracle/test/livehtml/sstest.html
Its virtual path is /sample/livehtml/sstest.html
```

On NT, you would see:

```
This document is \oracle\test\livehtml\sstest.html
Its virtual path is \sample\livehtml\sstest.html
```

Getting Information About Other Files

The fsize and flastmod commands allow you to get the file size and last modification date of any file on the server rather than just the current document.

For example, the following lines:

```
<!--#config sizefmt="bytes"-->
<p>This gives the file size of 'sstest.html' in bytes:
<!--#fsize file="sstest.html"-->
```

generate the following information:

```
This gives the file size of 'sstest.html' in bytes: 6405 bytes
```

The following lines:

```
<!--#config sizefmt="abbrev"-->
<p>This gives the file size of 'sstest.html' in bytes:
<!--#fsize file="sstest.html"-->
```

generate

```
This gives the file size of 'sstest.html' in kilobytes: 6 Kbytes
```

Displaying Browser Information

You can display the user's browser and version back to the user that he or she is using to read your pages.

The line:

```
<p>You are using <!--#echo var="HTTP_USER_AGENT" -->
```

produces the following:

```
You are using Mozilla/4.02 [en] (X11; I; SunOS 5.5.1 sun4u)
```

Providing Host and Server Information

The following line:

```
<p>Host: <!--#echo var="REMOTE_HOST" -->
(<!--#echo var="REMOTE_ADDR" -->)
<p>Server: <!--#echo var="SERVER_NAME" -->
(<!--#echo var="SERVER_SOFTWARE" -->)
```

generates the following information:

```
Host: test.us.oracle.com (123.45.67.89)
Server: test.us.oracle.com (Oracle_Web_Listener/4.0.6.3.0EnterpriseEdition)
```

Accessing a Database

The following command uses ICX to invoke a PL/SQL cartridge to run the **myproc** stored procedure. The results from the stored procedure are placed in the LiveHTML page. The example assumes that the **/db/plsql/** virtual path is associated with a PL/SQL cartridge.

```
<!--#request url="/db/plsql/myproc"-->
```

4

Writing Scripts

The scripting feature of the LiveHTML cartridge enables you to write embedded scripts and flow-control tags within HTML pages. Scripts can be inserted between standard HTML tags. These constructs allow you to perform more complex commands than those supported by Server-Side Includes (SSI). For example, you can write scripts to determine the user of your web application and use that user's name to determine the contents of the rest of the page. Flow-control tags ("if-then" statements) enable you to execute scripts or return HTML data on the page to the client only when the expression is true. For example, you can check the browser that the user is using, and return different HTML data depending on the browser type.

You use the scripting feature to access Web Application Objects from within LiveHTML pages. Web Application Objects provide a framework with runtime services for building transactional and non-transactional web applications. See Chapter 5, "Developing with Web Application Objects" for more information.

Currently, the cartridge runtime which has a Perl interpreter supports Perl version 5.004_01 as the scripting language. You can get the latest information about the Perl language from **http://www.perl.org**. You can also download many useful modules for the LiveHTML Perl interpreter from **http://www.perl.com/CPAN-local.modules/00modlist.long.html**. These modules can extend the functionality of your scripts. If you decide to upgrade the Perl interpreter, refer to Chapter 14, "Upgrading your Perl Interpreter".

Note: The LiveHTML Perl interpreter is database-enabled using the Perl DBD::Oracle driver and DBI API. See "Querying and Retrieving Data from an Oracle Database" on page 8 for a working example of querying an Oracle database.

Contents

Filename Extensions for Scripts

Pages for the LiveHTML cartridge that contain scripts are called HTML scripting pages (HSP). The default extensions set up for scripting pages are hsp, hsa, asp, and asa.

You can use other extensions if you add them to the Script Page Extension field in the LiveHTML Parameters configuration form in the Oracle Application Server Manager.

Enabling and Disabling the Scripting Feature

You can enable / disable the scripting feature. In the Cartridge Parameters page (in the Oracle Applications Server Manager) of a LiveHTML cartridge, check or uncheck the “Enable Script Execution” box.

If unchecked, scripts present in files with extensions specified under “Script Page Extension” are not processed by the cartridge. Those files are not parsed by the cartridge either. If the box is checked, scripts in those files are processed.

Specifying Scripting Languages

There are three ways of specifying scripting languages for LiveHTML pages. They differ in their scope of applicability.

Overall Default Language

The default scripting language for all pages belonging to a LiveHTML cartridge is specified in the cartridge configuration page in the Oracle Application Server Manager (see “Cartridge Configuration” in Chapter 2). This default language is applicable to all scripts unless an alternate language is specified using the methods detailed in the following two sections.

For a Particular Page

If you want to use another scripting language for a particular page, you can use the `<% @Language=`*`language`*` %>` tag to specify that language. The language specified applies to the scope of that page for scripts in the `<%...%>` and `<%=...%>` tags. If `<% @Language=`*`language`*` %>` is not present in a page, the default language specified in the cartridge configuration form applies to scripts inside the `<%...%>` and `<%=...%>` tags.

For a Script Block Within a Page

In certain situations, for a block of script, you may want to use a scripting language different from the default specified in Oracle Application Server Manager or in `<% @Language=`*`language`*` %>`. For these situations, use the `<SCRIPT>... </SCRIPT>` tag with its `LANGUAGE` attribute. Refer to the following section for more information.

Note: For the current release, only the "`Perl`" value is supported. More languages will be supported in future releases. Also, the Perl interpreter used in a LiveHTML cartridge server process is instantiated only once inside the process. To avoid unexpected results from your LiveHTML application, you should always initialize all the variables you use inside your LiveHTML scripts.

Embedding Scripts

There are several tags that allow you to embed scripts in LiveHTML pages. These are:

- <%...%>
- <%= ... %>
- <SCRIPT>...</SCRIPT>

The following table summarizes when to use each tag:

Table 4–1 Usage of scripting tags at glance

Tag	When to Use
<%...%>	Use when the scripts enclosed by this tag are written in the default language of the page.

Table 4–1 Usage of scripting tags at glance

Tag	When to Use
<%=...%>	Use when you need to output the result of an expression as part of the HTML output of a LiveHTML page.
<SCRIPT>...</SCRIPT>	Use when the script block enclosed by this tag is not written in the default language of the page. The language used in the script block is specific to this block alone.

Note: The results of scripts in the `<SCRIPT>...</SCRIPT>` or the `<%...%>` tags are seen by the user only if the script writes them through the write method of the Response object. For example, in Perl, the syntax is `$Response->write("text");`.

<%...%>

The `<%...%>` tag encloses scripts in the language specified by the `<% @Language = *language* %>` tag of each LiveHTML page. If this tag is not present in a page, the default scripting language specified in the cartridge configuration form is applicable for the scripts. You can interweave the standard HTML tags with this tag using control structures to determine which HTML tags are sent to the client. You can also embed these tags within other HTML tags. This can be useful for dynamically generated links (the HREF attribute of the A tag).

Syntax

```
<%
script
%>
```

Examples

The following example prints out the text “String to print” in incremental font sizes ranging from 3 to 7.

```
<%
  $str = "String to print";
  for ($fontsize = 3; $fontsize < 8; $fontsize++) {
%>
    <font size = <% $Response->write($fontsize) %> >
    <p> <% $Response->write($str) %> </font>
<% } %>
```

Note: The scope of variables is not limited to just one script block. Variables persist for script blocks over the entire page.

The following example shows how to use an if-then statement to determine which HTML statement to send to the client.

```
<%
  if ($something_failed) {
%>
<p>An error occurred while processing your request.
<% } else { %>
<p>Here are the results of your request.
<% } %>
```

<%= ... %>

The `<%= expression %>` tag displays the value of the specified expression which is written in the language specified in the `<% @Language=language %>` tag.

Syntax

```
<%= expression %>
```

Example

The following script prints “10 ... 9 ... 8 ... 7 ... 6 ... 5 ... 4 ... 3 ... 2 ... 1 ... Fire!”:

```
<% for ($countdown = 10; $countdown > 0; $countdown--) { %>
<%= $countdown %> ...
<% } %>
Fire!
```

<SCRIPT>...</SCRIPT>

This tag allows you to specify a specific language to use for the script block that it encloses. The language can be different from that specified in the `<% @Language=language %>` tag or in the cartridge configuration form.

Syntax

```
<SCRIPT
    RUNAT=SERVER
    [LANGUAGE=language]>
script
</SCRIPT>
```

Attributes

RUNAT - specifies that the tag is to be processed by the LiveHTML cartridge and not the client. This attribute is required for the LiveHTML cartridge to process the script. If not specified, the tag is processed by the client browser. The only allowed value currently is SERVER.

LANGUAGE - specifies the language of the script. Currently, “Perl” is the only allowed value. If this attribute is not specified, the default language specified in the cartridge configuration form is assumed.

Example

The following example displays a thank you message.

```
<SCRIPT RUNAT=SERVER LANGUAGE=Perl>
$Response->write("Thank you for using our <em>interactive web application
                 </em>.\n");
</SCRIPT>
```

Using CORBA Objects in Scripts

The <OBJECT> tag declares a CORBA object to be used in scripts in the current page. Multiple objects can be declared in the same page using this tag, but the scope of each declaration spans only that page. The object can then be referenced in scripts using the specified ID.

Syntax

```
<OBJECT
    RUNAT = "SERVER"
    ID= identifier
    OR = obj_ref_string | CARTRIDGE = cartridge_identifier>
```

Attributes

RUNAT - specifies that the tag is to be processed by the LiveHTML cartridge and not the client. This attribute is required for the LiveHTML cartridge to declare the object. If not specified, the tag is processed by the client browser. The only allowed value currently is SERVER.

ID - specifies the name by which the instance of the object can be referenced in scripts. This attribute is required.

One of the OR or CARTRIDGE attributes is required to locate the object.

OR - specifies the object reference in string form.

CARTRIDGE - specifies the name of the CORBA cartridge that defines and instantiates the object. The cartridge is managed by the Resource Manager.

Examples

The following example determines the value of the HREF attribute from the Menu object, which is defined in the <OBJECT> tag.

```
<OBJECT
    RUNAT = "SERVER"
    ID = "Menu"
    CARTRIDGE = "gui_set/menu_obj">
...
<A HREF="<%= $Menu->GetRef($date) %>">Daily Menu</A>
<!-- GetRef() is a method in the $Menu object -->
<!-- $date contains a date that the user could have entered earlier -->
```

The next example illustrates the use of the <OBJECT> tag to define three CORBA objects which are used in the embedded Perl script in a LiveHTML page. Each object is referenced using either OR or CARTRIDGE.

```
<!-- The embedded Perl scripts below retrieve balances from three accounts
belonging to the same person using three CORBA object "account managers". -->

<HTML>
<% $AccName = "Robert"; %>
<TITLE><%= $AccName %>'s Account Information</TITLE>
<BODY>

<!-- CORBA object referenced using object reference (for checking information).
-->
<OBJECT RUNAT="SERVER" ID="AccountManagerChecking"
OR="IOR:000000000000001C49444C3A42616E6B2F4163636F756E744D616E616765723A312E300
0000000010000000000000009C000101000000000A7270616E672D73756E00158800000008000100
009C82B05A0000000000000320000000000000000000031E33166329230000000000010059000000
00A7270616E672D73756E000000000000063132333839000000000000000000000050000001C4944
4C3A42616E6B2F4163636F756E744D616E616765723A312E30000000000A7069643A31323338390
000000000000000000000">
```

```
<!-- Checking account balance. -->
<% $AccountChecking = $AccountManagerChecking->open($AccName); %>
Checking account balance is $<%= $AccountChecking->balance() %><P>

<!-- CORBA object referenced using the name of the cartridge which will
instantiate it (CD account balance). -->
<object RUNAT="SERVER" ID="AccountManagerCD" Cartridge="AccountCApp/AccountC">

<!-- CD account balance. -->
<% $AccountCD = $AccountManagerCD->open($AccName); %>
CD account balance is $<%= $AccountCD->balance() %><P>

</BODY>
</html>
```

Generating IDL Interfaces of CORBA Objects for use in LiveHTML Pages

For the above example to work, IDL interfaces or client stubs for the CORBA object should already be generated so that the embedded scripts in the LiveHTML page can use the interfaces. To do this, the IDL-to-Perl compiler (perlidlc) that comes with Oracle Application Server is used. Sample command line usage of the compiler is shown below. For detailed information about using generated PERL bindings for IDL interfaces, please see Chapter 7, “Accessing CORBA Objects from Perl Scripts”.

```
prompt>  perlidlc -i -I $ORACLE_HOME/public -I $ORACLE_HOME/ows/apps/eco4j/
         MyAccount $ORACLE_HOME/ows/apps/eco4j/server/MyAccount/
         OASInterfaces.idl
```

where MyAccount is the name of your application and **OASInterfaces.idl** is the file containing the IDL source for an ECO / Java object.

For a non-JCORBA object, the command line will look like:

```
prompt>  perlidlc -i -I <include_directory> filename.idl
```

where *<include_directory>* contains include files specified in the IDL source **filename.idl**.

Scripting Examples

The following examples show how simple embedded scripts can be used. More examples can be found in the next few chapters as features are described.

Getting the Perl Version Number

The following script uses the `$]` variable to get the version of Perl used by the LiveHTML cartridge:

```
<p>The LiveHTML cartridge uses Perl version
<%= $] %>.
```

Invoking a Function in an Included Perl Module

The following script invokes the `ctime` function in the `Time::localtime` module:

```
<p>The local time is:
<% use Time::localtime; %>
<%= ctime(time()) %>
```

5

Developing with Web Application Objects

Oracle Application Server provides a versatile and extensible set of objects that you can access from your LiveHTML scripts. This set of objects form a basic framework that provides runtime services required to build transactional web-based applications. The objects allow your scripts to query and modify the execution or runtime environment of a LiveHTML page. Using them, your scripts can perform tasks such as obtaining HTTP header information, retrieving and setting cookies, creating or accessing CORBA-compliant objects, making requests to other cartridges, committing or rolling-back transactions. (For information about transactional Web Application Objects, refer to Chapter 6, “Transactions in LiveHTML”).

With Web Application Objects, you can extend the functionality of your applications by allowing complex custom logic to be developed as external reusable components (e.g. ECO / Java and Enterprise Java Beans objects, JWeb and PL / SQL cartridges). These components can be used in your web applications by invoking them from your LiveHTML scripts.

Contents

What are Web Application Objects

Web Application Objects provide a set of objects for you to interact with the runtime environment of web applications. They are implemented as CORBA objects with IDL interfaces, and they encapsulate lower level operations so that you can focus on developing content with enhanced functionality provided by Oracle Application Server.

Usage examples of Web Application Objects are operations such as retrieving HTTP header values, setting cookies, controlling the transactional attributes of a page, accessing objects and methods in other cartridges.

The following table gives a summary of the currently available Web Application Objects.

Table 5–1 Overview of Web Application Objects

Object Set	Object type	Description
Core	Request	The HTTP request sent by the user.
	Response	The server's response to the user's request.
	HTTPListener	The web listener that received the request.
	Server	For creating objects and managing the object factory.
	Document	An object representing the LiveHTML page it is referenced in.
	ObjectFactory	For creating or retrieving objects in the Oracle Application Server environment.
Collection / Container	Vector	A dynamically resizable array utility.
	Iterator	An object to move between elements in containers.
	Hashtable	A utility for retrieving name-value pairs quickly.
I/O	HTTPInputStream	Stream for reading data sent by the client.
	HTTPOutputStream	Stream for writing data to the client.
ICX (inter-cartridge exchange)	ICXRequest	For making inter-cartridge requests in the Oracle Application Server. Uses URL addressing to target a cartridge. This object supports transactional context propagation.
	ICXResponse	For retrieving data from other cartridges. This object supports transactional context propagation.
Utility	Cookie	For setting and getting cookies.
Transaction	TxContent	To commit or rollback a transaction and obtain transactional status information.

These objects have methods and attributes which you can use to define their characteristics. To use these objects, you have to invoke their methods, and read or change their attributes.

Scripting with Web Application Objects

Web Application Objects have attributes and methods that you can use for developing your web application. You access the objects through scripts embedded in your LiveHTML page. (See the “Embedding Scripts” on page 4-3 section for scripting details.) The syntax for referencing the objects and their methods and attributes follows the syntax of the scripting language. Currently, the supported language is Perl and our discussion will center around that.

Using Perl

For scripts written in Perl, you can reference all the objects’ methods and attributes through the following five pre-defined variables:

```
$Server
$Document
$Request
$Response
$TxContext (detailed in Chapter 6, “Transactions in LiveHTML”)
```

These five variables are created automatically; you do not have to declare or initialize them in any way before you can use them. Also, Perl is case-sensitive; note that the first letter of the object name is in uppercase.

From a scripting syntax perspective, methods and attributes of Web Application Objects can be invoked or accessed directly using one of the appropriate variables and the arrow (->) operator without having to specify the owner objects. For example, to invoke the write() method of the OutputStream object, the syntax is:

```
$Response->write("Hello World\n");
```

The following table illustrates the groupings of Web Application Objects under the five Perl variables:

$Server	$Request	$Response	$Document	$TxContext
Server ObjectFactory	Request HTTPListener InputStream	Response OutputStream	Document	TxContext (Refer to Chapter 6, “Transactions in LiveHTML”)

Hence, to retrieve the svr_name attribute of the Request object, you can use the `$Request` variable:

```
$sname = $Request->svr_name();
```

Note: Some attributes are read-only and cannot be set.

The remaining Web Application Objects fall into two groups: collections and utility / ICX objects:

Collection Objects	Utility/ICX Objects
Vector	Cookie (in package oracle::OAS::WAO::HTTP::Cookie)
Iterator	ICXRequest (in package oracle::OAS::WAO::OASFrmkObject)
HashTable	ICXResponse (in package oracle::OAS::WAO::OASFrmkObject)

Collection objects consist of the HashTable (key-value) and Vector (dynamic array) container objects and the Iterator object which performs operations on the latter two. These objects are useful for data manipulation. To access methods and attributes of the container objects, you need to use a variable which contains a reference to either object. For example, if you want to retrieve HTTP header information into HashTable, you can use the get_headers() method of the Request object to obtain a reference to the headers. This reference can then be used by the Iterator object's methods to manipulate the header data. The following script illustrates:

```
$headers = $Request->get_headers(); # obtain a reference for the headers
$keylist = $headers->keys(); # obtain a reference for the keys of the header hash
$key = $keylist->get_next_element()->extract();
$value = $headers->get_value($key)->extract(); # value corresponding to key name is
                                                 retrieved
```

Note: The extract() method is needed because the return type of header information is of the IDL Any type which doesn't map to any Perl types.

Utility / ICX objects provide added functionality to the web application object framework. Each of them has specific functions. Examples of use are in inter-cartridge exchanges and cookie operations. To use a utility / ICX object in Perl, you need to specify the package name that contains it. You also need to obtain an object reference using the get_object() method of the Server object. The following script performs these actions for a Cookie object (applies to ICX objects also):

```
# Load the Perl package containing the Cookie object
use oracle::OAS::WAO::HTTP::Cookie;
$cookie = oracle::OAS::WAO::HTTP::Cookie->narrow($Server->get_object("wao://Cookie"));
# the narrow method is used to narrow the generic CORBA object returned by
# Server.get_object to the correct interface
```

The following section provides examples to illustrate scripting with Web Application Objects.

Examples

The examples are written in the Perl language. To use these scripts, you need to place them within the <% ... %> tags.

Using the Response Object to Return Data to the Client

The following example uses the `sout` attribute of the Response object to get the output stream object, and then calls the output stream's `write_wstring()` method to return string data to the client.

```
<%
for ($i = 1; $i <= 5; $i += 2) {
  $Response->sout->write_wstring("The value of i is $i \n");
}
%>
```

The output of this script is:

```
The value of i is 1
The value of i is 3
The value of i is 5
```

Using the write() method of the Response object produces the same result. That is,

```
$Response->write("The value of i is $i \n");
```

The latter syntax is recommended as it invokes the same operation with shorter syntax than the first.

Displaying HTTP Headers

Using the Request object, you can display the HTTP headers in a request. The script looks like:

```
<%
# Load the package for HTTP request
use oracle::OAS::WAO::HTTP::HTTPRequest;
```

```
$headerhash = $Request->get_headers();   # headerhash is a hashtable
# cycle through the contents of the headerhash hashtable
$keylist = $headerhash->keys();          # keylist is an Iterator
while ($keylist->has_more_elements()) {
    $a_key = $keylist->get_next_element();    # get the next key
    $val = $headerhash->get_value($a_key);    # get the value for the key
    $Response->write("$a_key = $val \n");
}
%>
```

Note that this example provides an idea of how to use the Request, Iterator, and Hashtable objects.

Getting the Physical Path from a Virtual Path

The following example gets the corresponding physical path from a virtual path.

```
<%
$listener = $Request->listener();
$ppath = $listener->map_path("/testpages/index.html");
%>
```

Setting a Cookie

The Response object can be used for sending cookies to clients. This is illustrated in the following example:

```
<%
# Load the package for Cookie
use oracle::OAS::WAO::HTTP::Cookie;

# Utilize the predefined variable $Server to instantiate a
# cookie object.  Since $Server->get_object returns a generic
# CORBA object, you need to narrow it to the right interface,
# i.e. oracle::OAS::WAO::HTTP::Cookie. That will give you an empty cookie.
my $cookie = oracle::OAS::WAO::HTTP::Cookie->narrow($Server->
                                                     get_object("wao://Cookie"));

# set the various attributes of the cookie like name, value, domain,...
$cookie->name("user");
$cookie->value("john");
$cookie->path("");
$cookie->domain("www.foo.com");
$cookie->expires("31-DEC-99 GMT");
```

```
# Finally, set the cookie in the "Response" object
$value = $Response->set_cookie($cookie);
%>
```

Retrieving a Cookie

```
<%
# Load the package for Cookie and create an object reference to Cookie
use oracle::OAS::WAO::HTTP::Cookie;
my $cookie = oracle::OAS::WAO::HTTP::Cookie->narrow($Server->
                                                    get_object("wao://Cookie"));

# Retrieve our cookie "user"
my $cookie = $Request->get_cookie("user");
use Oracle::hlpr::Excp;
# If there is no exception raised when retrieving the cookie, implying cookie
# is set in client's browser, the cookie value is valid.
if (!Oracle::hlpr::Excp->isexcp($cookie)) {
    $cookie = $cookie->extract();
}
%>
```

Invoking a PL/SQL Procedure

```
<%
# Load the package for ICXRequest and create an object reference to ICXRequest
use oracle::OAS::WAO::OASFrmkObject;
my $icx_object = oracle::OAS::WAO::OASFrmkObject->narrow($Server->
                                                get_object("wao://ICXRequest"));

# initialize the object with the URL address of the target
$icx_object->init = ("http://machine/plsqlapp/cartx/procedure_name");

# set the Http request method to GET
$icx_object->set_method(GET);

# create a new ICX connection
$resp_object = $icx_object->connect();

# grab the result of the procedure
$foo = $resp_object->content();
...
%>
```

Querying and Retrieving Data from an Oracle Database

```
<!-- The following script logs on to an Oracle database, queries for data and
retrieves them; the Perl DBD::Oracle driver is used with the DBI API. HTML
formatting tags are included for completeness -->

<HTML>
<TITLE>OAS LiveHTML Scripting Example</TITLE>
<BODY BGCOLOR="#FFFFFF">
<%
# this subroutine sets up the DB connection
sub set_connection {
    my ($cs, $name, $pwd)=@_;
    use DBI;
    $logon = DBI->connect($cs, $name, $pwd,"Oracle") ||
                          $Response->write("Failed to logon: $DBI::errstr\n");
}

# this subroutine returns the values of a query
sub get_users {
    my($sqlstmt) = @_;
    $query=$logon->prepare("select username from all_users") ||
                            $Response->write("Failed to perform query
                                              $DBI::errstr\n");
    $query->execute() ||
            $Response->write("Failed to perform query $DBI::errstr\n");
    my $i=0;
    my @emp;
    while(@fields = $query->fetchrow()){
        # need to add the row to an array
        @emp->[$i]=@fields[0];
        $i++;
    }
    return @emp;
}

sub get_user_details {
    my ($username) = @_;
    $query=$logon->prepare("select * from all_users
                            where username = '$username'") ||
                            $Response->write("Failed to perform query
                                              $DBI::errstr\n");
    $query->execute() ||
    $Response->write("Failed to perform query $DBI::errstr\n");
    my @fields=$query->fetchrow();
    return @fields;
```

```
}
%>

<H2>Server Side Scripting Demonstration</H2>
<P>
<HR ALIGN="LEFT" WIDTH="75%" SIZE="5">
</P>

<%
# Main

# determine if form data was sent
my $formvalue=$Request->form();
if($formvalue){
    my $username=$formvalue->get_value("username")->extract();
    my @details=get_user_details($username);
%>
    <!-- display user details -->
    <!-- outer table -->
    <TABLE BORDER="2" CELLPADDING="10" CELLSPACING="0" WIDTH="75%">
    <TR>
    <TD WIDTH="33%" BGCOLOR="#FFFFCC">USERNAME</TD>
    <TD WIDTH="33%" BGCOLOR="#FFFFCC">USER_ID</TD>
    <TD WIDTH="33%" BGCOLOR="#FFFFCC">CREATED</TD>
    </TR>
    <TR>
    <%
    my $i=0;
    for(@details){
    %>
    <TD WIDTH="33%" BGCOLOR="#66CCCC"><%=@details[$i++]%></TD>
    </TR>
    </TABLE>

<% } else {%>

    <TABLE BORDER="2" CELLPADDING="10" CELLSPACING="0" WIDTH="75%" HEIGHT="198">
    <TR>
    <TD WIDTH="100%" BGCOLOR="#FFFFCC">Please select a User from below</TD>
    </TR>
    <TR>
    <TD WIDTH="100%">
    <P>
    <!-- inner table -->
    <FORM ACTION="topics.hsp">
    <INPUT TYPE="HIDDEN" NAME="GETDETAILS" VALUE="TRUE">
    <TABLE BORDER="0" WIDTH="100%" CELLSPACING="5" CELLPADDING="5">
    <!-- add each of the users here -->
    <%
    set_connection("machine_name","scott","tiger");
    my @users=get_users();
    my $i=0;

    for(@users){
    %>
        <TR>
        <TD WIDTH="18%" BGCOLOR="#66CCCC">
        <input type="radio" name="username" value="<%=@users[$i]%>"></TD>
        <TD WIDTH="82%" BGCOLOR="#66CCCC"><%=@users[$i++]%></TD>
        </TR>
    <% } %> <!-- for loop -->
    <TR>
    <!-- the padding for the buttons -->
    <TD WIDTH="18%">&nbsp;</TD>
    <TD WIDTH="82%">&nbsp;</TD>
    </TR>
    <TR>
    <!-- the form submit buttons -->
    <TD ALIGN="RIGHT" WIDTH="18%">&nbsp;</TD>
    <TD ALIGN="RIGHT" WIDTH="82%">
    <INPUT TYPE="SUBMIT" VALUE="Show User Details">
    <INPUT TYPE="RESET">
    </TD>
    </TR>
    </TABLE> <!-- inner-->
    </FORM>
    </P>
    </TD>
    </TR>
    </TABLE> <!-- outer -->
<% } %> <!-- if statement -->
</BODY>
</HTML>
```

Summary of Methods and Attributes

The following table lists the attributes and methods that you can use for each web application object:

Table 5–2 List of methods and attributes

Object	Methods	Attributes

Request	get_header()
	get_headers()
	get_cookie()
	get_cookies()
	get_request_info()
	get_cgivar()
	init()
	destroy()
		id
		protocol
		protocol_major_ver
		protocol_minor_ver
		svr_name
		svr_port
		form
		query_string_vars
		cgivars
		sin
		content_len
		content_type
		listener
		auth_type
		method

Response	set_header()
	set_headers()
	set_cookie()
	set_cookies()
	end()
	redirect()
	set_status()
	set_status_and_msg()
	set_expire_length()
	set_expire_date()
	set_content_len()
	set_content_type()
	send_headers()
	write()
	init()
	destroy()
		id
		sout
		keep_alive
		auto_send
		headers_sent

HTTPListener	map_path()	host_name port

Table 5–2 List of methods and attributes

Object	Methods	Attributes
Cookie		name value path domain expire secure

OutputStream	close()
	flush()
	write()
	write_wstring()
	write_wchars()	

InputStream	ready()
	close()
	read()
	read_wchars()
	mark()
	reset()
	skip()
		mark_supported

Vector	expand()
	add_element()
	add_elements()
	get_element()
	get_elements()
	set_element()
	set_elements()
	insert_element()
	insert_elements()
	remove_element()
	remove_elements()
	elements()
	lock()
	unlock()
	init()
	destroy()
		capacity
		num_elements
		capacity_increment
		auto_resize

Table 5–2 List of methods and attributes

Object	Methods	Attributes

Iterator	has_more_elements()
	get_next_element()
	get_next_elements()
	skip()
	reset()
	clone_iterator()
	init()
	destroy()	

Hashtable	clear()
	aggregate()
	get_value()
	get_values()
	set_value()
	set_values()
	remove_value()
	remove_values()
	keys()
	lock()
	unlock()
	init()
	destroy()
		num_keys

ICXRequest	set_method()
	set_header()
	set_headers()
	set_content()
	set_contents()
	set_auth_info()
	connect()
	enable_transaction()
	disable_transaction()
	init()
	destroy()	

ICXResponse	get_header()
	get_headers()
	init()
	destroy()
		status_code
		realm
		reason_phrase
		http_version
		using_proxy
		content

ObjectFactory	get_object()	

Table 5–2 List of methods and attributes

Object	Methods	Attributes

Server	`get_object_factory()`
	`set_object_factory()`
	`get_object()`
	`get_object_by_type()`
	`init()`
	`destroy()`	

Document		`tx_attr` `dft_script_language`

Request

An instance of the Request object is created when a client sends a request to a LiveHTML cartridge. This object can be used for extracting client request information (e.g. HTTP header or query string information).

Table 5–3 Request methods

Method	Syntax and Description

get_header()	`wstring get_header(in wstring header_name)`
	`raises (oracle::OAS::Util::NoSuchElement)`
	Returns the value of the specified HTTP header *header_name*.

get_headers()	`oracle::OAS::Util::Hashtable get_headers()` Returns all HTTP headers as a Hashtable.

get_cookie()	`any get_cookie(in wstring cookie_name)`
	`raises (oracle::OAS::Util::NoSuchElement)`
	Returns the value of the specified cookie *cookie_name*.

get_cookies()	`oracle::OAS::Util::Hashtable get_cookies()` Returns all cookies sent by the client.

Table 5–3 Request methods

Method	Syntax and Description
get_request_info()	`wstring get_request_info(in HTTPRequestInfoType` *`type`*`)` Returns information associated with the request. *`type`* is one of: `uri` - the request URI `url` - the request URL `listener_type` - the type and version of the listener `virtual_path` - the virtual path for the request `physical_path` - the physical path for the request `query_string` - the query string `language` - comma-delimited list of languages `encoding` - comma-delimited list of encodings `mime_type` - MIME type `user_id` - user ID `password` - password `ip_addr` - IP address in a.b.c.d notation
get_cgivar()	`wstring get_cgivar(in wstring` *`name`*`)` Returns the value of the specified CGI environment variable.
init()	`void init(in any` *`init_param`*`)` Initializes this object. *`init_param`* is optional.
destroy()	`void destroy()` Destroys this object.

Table 5–4 Request attributes

Attribute	Read or Read/Write	Type	Description
id	R	long	The identifier for this instance of the Request object.
protocol	R	string	The protocol of the request.
protocol_major_ver	R	long	The major version of the request protocol.

Table 5–4 Request attributes

Attribute	Read or Read/Write	Type	Description
protocol_minor_ver	R	long	The minor version of the request protocol.
svr_name	R	wstring	The name of the server that received this request.
svr_port	R	long	The port number on which this request is received.
form	R	oracle::OAS::Util::Hashtable	The values of form elements in the HTTP request body returned in a hashtable.
query_string_vars	R	oracle::OAS::Util::Hashtable	The values of variables in the HTTP query string returned in a hashtable.
cgivars	R	oracle::OAS::Util::Hashtable	Retrieves CGI 1.1 variables associated with the request and returns them in a hashtable.
sin	R	oracle::OAS::IO::InputStream	An instance of the InputStream object representing the request entity data from the client.
content_len	R	long	The size of the request entity data in bytes.
content_type	R	string	The MIME type of the request entity data.
listener	R	HTTPListener	The HTTP listener with which this request is associated.
auth_type	R	string	The authentication scheme of the request. This can be one of the standard schemes (basic, oracle, ...) or a custom scheme.
method	R	string	The method of the request. Possible values are “HEAD”, “GET”, or “POST”.

Response

The LiveHTML cartridge creates an instance of the Response object and sends it to the client in response to a request. This object can be used to send data to the client.

Table 5–5 Response methods

Method	Syntax and Description
set_header()	`void set_header(in string `*`name`*`, in string `*`value`*`)` Adds a new HTTP header using the *name* and *value* pair specified.
set_headers()	`void set_headers(in oracle::OAS::Util::Hashtable `*`headers`*`)` Adds HTTP headers contained in the hashtable.
set_status()	`void set_status(in long `*`status_code`*`)` Sets the status code to be returned.
set_status_and_msg()	`void set_status_and_msg(in long `*`status_code`*`, in wstring `*`status_msg`*`)` Sets the status code and status message to be returned.
set_expire_length()	`void set_expire_length(in long `*`expire_length`*`)` Specifies how long before a cached page on the browser expires.
set_expire_date()	`void set_expire_date(in oracle::OAS::Util::Date `*`expire_date`*`)` Specifies the date and time on which a page cached on the browser expires.
set_content_len()	`void set_content_len(in long `*`content_len`*`)` Sets the content length of the response.
set_content_type()	`void set_content_type(in wstring `*`content_type`*`)` Sets the content type of the response.
set_cookie()	`void set_cookie(in oracle::OAS::WAO::HTTP::Cookie `*`cookie`*`)` Sets a cookie to be sent back to client.
set_cookies()	`void set_cookies(in oracle::OAS::Util::Hashtable `*`cookies`*`)` Sets cookies to be sent back to the client.
send_headers()	`void send_headers()` Send all the headers that have been set so far to the client.
write()	`void write(in wstring `*`msg`*`)` Sends a string to the client.

Table 5–5 Response methods

Method	Syntax and Description
end()	`void end()` Stops processing and returns the result to the client. This method calls `flush()` on the OutputStream.
redirect()	`void redirect(in string` *`new_url`*`)` Sends a redirect message to the browser, causing it to attempt to connect to the specified URL.
init()	`void init(in any` *`init_param`*`)` Initializes this object. *`init_param`* is optional.
destroy()	`void destroy()` Destroys this object.

Table 5–6 Response attributes

Attribute	Read or Read/Write	Type	Description
id	R	long	The identifier for this instance of the object.
sout	R	oracle::OAS::IO::OutputStream	Response output stream.
auto_send	R/W	boolean	If `true`, headers are sent after each call that modifies the response header.
headers_sent	R	boolean	If `true`, headers have already been sent.
keep_alive	R/W	boolean	The value is TRUE if HTTP 1.1 KeepAlive is enabled

Cookie

The Cookie object is implemented using the attributes shown in the table below. The object complies to the Netscape 2.0 specification for client cookies. It can be used to store state information in the client browser for subsequent reuse in future HTTP requests.

Table 5–7 Cookie attributes

Attribute	Read or Read/Write	Type	Description
name	R/W	wstring	The name of this cookie.
value	R/W	wstring	The value of this cookie.
path	R/W	wstring	The URL for which this cookie is presented. If the URL does not begin with this path, the cookie is not presented.
domain	R/W	wstring	The domain of this cookie.
expires	R/W	wstring	The expire time of this cookie. If this value is not specified, the cookie will be discarded when the client quits the session.
secure	R/W	boolean	The value of the secure flag. The default is false.

HTTPListener

This object represents a listener that is managed by Oracle Application Server. It can be used to retrieve information about the listener, e.g. port number.

Table 5–8 HTTPListener methods

Method	Syntax and Description
map_path()	`string map_path(in string *vpath*)` Returns the corresponding physical path for the specified virtual path.

Table 5–9 HTTPListener attributes

Attribute	Read or Read/Write	Type	Description
host_name	R	string	The name of the machine on which the listener is running.
port	R	long	The port number at which the listener is listening.

OutputStream

The output stream enables you to send data to the client. Typically, you would access an output stream object from the `sout` attribute of the Response object. For example:

```
$Response->sout->write_wstring("Send this line back to the client\n");
```

This produces the same result as:

```
$Response->write("Send this line back to the client\n”);
```

Note: Multibyte characters are not supported in streams.

Table 5–10 OutputStream methods

Method	Syntax and Description
close()	`void close()` Closes the output stream.
flush()	`void flush()` Flushes the contents of the buffer to be written.
write()	`void write(in wchar` *`c`*`)` Writes one character to the output stream. This is different from the write() method of the Response object which handles strings.
write_wstring()	`void write_wstring(in wstring` *`s`*`)` Writes one or more characters to the output stream. This is similar to the write() method of the Response object.
write_wchars()	`void write_wchars(in wcharSeq` *`cSeq`*`)` Writes an array of characters to the output stream.

InputStream

Input streams allow you to read data from clients. You can get an instance of an input stream from the `sin` attribute of the Request object.

Note: Multibyte characters are not supported in streams.

Table 5–11 InputStream methods

Method	Syntax and Description
ready()	`boolean ready()` Whether or not the input stream for the application is ready. Each application can have only one input stream.
close()	`void close()` Closes the input stream.
read()	`wchar read()` Gets a character from the input stream.
read_wchars()	`long read_wchars(inout wcharSeq` *`cSeq`*`)` Gets an array of characters from the input stream.
mark()	`void mark(in long` *`readLimit`*`)` Place a mark at the specified location.
reset()	`void reset()` Move the current position to the mark.
skip()	`long slip(in long` *`nChars`*`)` Skips the specified number of characters and discards them. Return value is the number of characters skipped.

Table 5–12 InputStream attributes

Attribute	Read or Read/Write	Type	Description
mark_supported	R	boolean	Whether or not the input stream supports marks. Marks work like bookmarks: you mark a place in the stream, and return to it anytime later.

Vector

A vector behaves like a dynamically resizable array. The first index in a vector is `0`.

Table 5–13 Vector attributes

Attribute	Read or Read/Write	Type	Description
capacity	R/W	long	The current size of the vector. If you decrease the size, you could lose some existing elements in the vector.
num_elements	R/W	long	The number of elements in the vector.
capacity_increment	R/W	long	The amount by which the vector grows. The vector grows when you add elements beyond its capacity. The default value is -1, which enables the vector to choose an optimal value for resizing.
auto_resize	R/W	boolean	If true, the vector automatically increases its capacity to accommodate out-of-bounds indexes. If false (the default), the vector throws an ArrayOutOfBounds exception when an out-of-bounds index is detected.

Table 5–14 Vector methods

Method	Syntax and Description
expand()	`void expand()` Increases the size of the vector by the amount specified in `capacity_increment`.
add_element()	`void add_element(in any element)` Adds the specified element to the vector. The vector resizes automatically, if necessary, to fit the element.
add_elements()	`void add_elements(in anySeq elements)` Adds the specified elements to the vector. The vector resizes automatically, if necessary, to fit the elements.

Table 5–14 Vector methods

Method	Syntax and Description
get_element()	`any get_element(in long index)` Returns the element at the specified index.
get_elements()	`anySeq get_elements(in long begin_index, in long num_elements)` Returns *num_elements* elements starting at *begin_index*.
set_element()	`void set_element(in long index, in any element)` Sets the element at the specified index, overwriting the content of the index. Note that if `auto_resize` is `false` and index is greater than the capacity of the vector, an ArrayOutOfBounds exception is thrown.
set_elements()	`void set_elements(in long begin_index, in anySeq elements)` Sets the elements starting at *begin_index*. Any elements in those indexes will be overwritten. Note that if `auto_resize` is `false` and index increases to values greater than the capacity of the vector, an ArrayOutOfBounds exception is thrown.
insert_element()	`void insert_element(in long index, in any element)` Inserts an element at the specified index, and the previous elements at the index and above will be bumped up one position. Note that if `auto_resize` is `false` and index is greater than the capacity of the vector, an ArrayOutOfBounds exception is thrown.
insert_elements()	`void insert_elements(in long begin_index, in anySeq elements)` Inserts elements starting at *begin_index*, and the previous elements at those positions will be bumped up. Note that if `auto_resize` is `false` and index is greater than the capacity of the vector, an ArrayOutOfBounds exception is thrown.
remove_element()	`void remove_element(in long index)` Removes the element at the specified index, and elements at indexes greater than the specified index are shifted down by one. Note that removing an element just removes it from the vector; it does not destroy the element itself. If an element does not exist at the specified index, a NoSuchElement exception is thrown.
remove_elements()	`void remove_elements(in long begin_index, in long num_elements)` Removes the elements starting at *begin_index*, for *num_elements*. All elements at indexes greater than (begin_index + num_elements) are shifted down.
elements()	`Iterator elements()` Creates a new iterator that you can use to traverse the vector to get its contents. To ensure that the iterator remains valid, you should lock the vector before creating the iterator and unlock it after destroying the iterator. Ensure that you do not need to modify the vector while it is locked.

Table 5–14 Vector methods

Method	Syntax and Description
lock()	`void lock()` Locks the vector so that its contents cannot be modified.
unlock()	`void unlock()` Unlocks the vector.
init()	`void init(in any init_param)` Initializes this object. `init_param` is optional.
destroy()	`void destroy()` Destroys this object.

Iterator

This is an object to move between elements in containers. An instance of this object is created for each series of elements. When created, it is positioned before the first element of the series.

Table 5–15 Iterator methods

Method	Syntax and Description
has_more_elements()	`boolean has_more_elements()` Returns true if the series contains more elements.
get_next_element()	`any get_next_element()` Returns the next element and moves the iterator to the following element.
get_next_elements()	`anySeq get_next_elements(in long n)` Returns the next n elements in the series and advances the iterator n positions.
skip()	`void skip(in long n)` Skips the next n elements.
reset()	`void reset()` Brings the iterator's position back to the first element, if one exists.
clone_iterator()	`Iterator clone_iterator()` Creates a new iterator instance at the same position as the existing one.
init()	`void init(in any init_param)` Initializes this object. `init_param` is optional.
destroy()	`void destroy()` Destroys this object.

Hashtable

A hashtable is a mechanism that enables you to store and retrieve name-value pairs quickly.

Table 5–16 Hashtable attributes

Attribute	Read or Read/Write	Type	Description
num_keys	R	long	The number of keys in the hashtable.

Table 5–17 Hashtable methods

Method	Syntax and Description
clear()	`void clear()` Removes all the keys from the hashtable.
aggregate()	`void aggregate(in Hashtable table)` Adds the contents of the specified hashtable into this hashtable.
get_value()	`any get_value(in wstring key)` Returns the value associated with the specified key.
get_values()	`anySeq get_values(in wstringSeq keys)` Returns the values associated with the specified keys.
set_value()	`void set_value(in wstring key, in any value)` Adds a key-value pair to the hashtable.
set_values()	`void set_values(in wstringSeq keys, in anySeq values)` Adds multiple key-value pairs to the hashtable.
remove_value()	`void remove_value(in wstring key)` Removes the key and its value from the hashtable.
remove_values()	`void remove_values(in wstringSeq keys)` Removes the specified keys and their values from the hashtable.

Table 5–17 Hashtable methods

Method	Syntax and Description
keys()	`Iterator keys()` Returns an iterator that you can traverse to get the keys in the hashtable. To ensure that the iterator remains valid, you should lock the hashtable before creating the iterator and unlock it after destroying the iterator. Ensure that you do not need to modify the hashtable while it is locked.
lock()	`void lock()` Locks the hashtable so that it cannot be modified.
unlock()	`void unlock()` Unlocks the hashtable.
init()	`void init(in any init_param)` Initializes this object. `init_param` is optional.
destroy()	`void destroy()` Destroys this object.

ICXRequest

The ICXRequest object is used to make requests from the LiveHTML cartridge to other cartridges in Oracle Application Server.

Table 5–18 ICXRequest methods

Method	Syntax and Description
set_method()	`void set_method(in HTTPRequestMethod method)` Sets the method for this request. `method` can be any of the following as specified in the HTTP 1.1 specifications: GET, HEAD, POST, PUT, DELETE, TRACE.
set_header()	`void set_header(in wstring name, in wstring value)` Sets a single header for this request where `name` is the name of the header and `value` is the value of the header.
set_headers()	`void set_headers(in oracle::OAS::Util::Hashtable headers)` Appends specified headers to the existing set of headers where `headers` is a hashtable containing headers to be appended to the existing set of headers. Note that the contents of the hashtable can be of the following types only: wstring, Cookie, or Vector. If Vector type, then each entry of the Vector must be either a string or Cookie.
set_content()	`void set_content(in wstring name, in wstring value)` Sets the content for the request. `name` is the name of the content and value is the value of the content.

set_contents()	`void set_contents(in oracle::OAS::Util::Hashtable contents)`
	`Appends the contents in a hashtable to the existing set of contents in the request. contents is the hashtable containing the new items.`

set_auth_info()	`void set_auth_info(in wstring name, in wstring password)`
	`Sets the username and password for this request.`

connect()	`ICXResponse connect()`
	`Establishes an ICX connection and returns an ICXResponse object representing a new ICX connection.`

enable_transaction()	`void enable_transaction()`
	`Propagate transactional context with the current ICX request.`

disable_transaction()	`void disable_transaction()`
	`Disable transactional context propagation with the current ICX request.`

Table 5–18 ICXRequest methods

Method	Syntax and Description
init()	`void init(in wstring url)` Sets the URL location of the target object where `url` is the URL address.
destroy()	`void destroy()` Destroys this object.

ICXResponse

The ICXResponse object is used to obtain replies from other cartridges responding to a ICXRequest.

Table 5–19 ICXResponse methods

Method	Syntax and Description
get_header()	`wstring get_header(in wstring header_name)` Obtains the header value associated with the specified `header_name`.
get_headers()	`oracle::OAS::Util::Hashtable get_headers()` Obtains the values of all headers associated with this ICXResponse object in a hashtable.
init()	`void init(in any init_param)` Initializes this object. `init_param` is optional.
destroy()	`void destroy()` Destroys this object.

Table 5–20 ICXResponse attributes

Attribute	Read or Read/Write	Type	Description
status_code	R	long	Specifies the HTTP response code.
realm	R	wstring	Specifies the name of the authentication realm specified in the response.
reason_phrase	R	wstring	Specifies the reason text string that corresponds to the HTTP response code.
http_version	R	wstring	The version of the HTTP being used.
using_proxy	R	boolean	Indicates whether a proxy was used in the request.
content	R	oracle::OAS::IO::InputStream	Read the content as input stream (text-based).

ObjectFactory

This object creates a new object or retrieves an existing object.

Table 5–21 ObjectFactory methods

Method	Syntax and Description
get_object()	`Object get_object(in wstring obj_name)` Retrieves an existing object or creates a new one according to the factory-specific name specified in `obj_name`. For example, for WAOObjectFactory, one of the default factories provided by Oracle Application Server, the following values are valid for `obj_name`: Vector, Hashtable, Cookie, and ICXRequest.

Server

The Server object represents the runtime environment for the Oracle Application Server. It is used to provide object creation capabilities.

Table 5–22 Server methods

Method	Syntax and Description
get_object_factory()	`ObjectFactory get_object_factory(in wstring obj_factory_type)` Get the object factory for the specified object type. Valid values for `obj_factory_type` are: “wao” - representing the factory for Web Application Objects “ior” - representing the factory for objects specified by their stringed CORBA object reference “cartx” - representing the factory for an Oracle Application Server cartridge object
get_object()	`Object get_object(in wstring name)` Retrieves or creates an object using the given name. Returns an object reference representing the requested object or a null value if the object is not found or cannot be created. Note that the actual `name` must be prefixed with one of the following: “wao://” - representing the WAO Framework Object Set (any web application object) “ior://” - a stringed CORBA object reference “cartx://” - the name of an Oracle Application Server cartridge
get_object_by_type()	`Object get_object_by_type (in wstring type, in wstring name)` Retrieves or creates an object based on the object’s type and name. Returns an object reference representing the requested object or a null value if the object is not found or cannot be created. `name` must be one of the following: “wao” - WAO Framework Object Set “ior” - a stringed CORBA object reference “cartx” - the name of an Oracle Application Server cartridge
init()	`void init(in any init_param)` Initializes this object. `init_param` is optional.
destroy()	`void destroy()` Destroys this object.

Document

The Document object represents each LiveHTML page it is instantiated in. It allows you to access attributes of each page.

Table 5–23 Document attributes

Attribute	Read or Read/Write	Type	Description
tx_attr	R	string	Returns a value indicating the transactional context of the current page. 0 = must participate in transaction 1 = starts a new transaction 2 = in transaction 3 = not in transaction
dft_script_language	R	wstring	Returns a string indicating the default script language of the current page. Valid value is PERL for current release.

6

Transactions in LiveHTML

You can develop applications with transactional features using Web Application Objects. These features are implemented on a per-page basis for each LiveHTML page that has the transaction property enabled. A transaction can span more than one cartridge provided each cartridge involved has transaction support enabled. See the Oracle Application Server Administration Guide.

Contents

Specifying the Transactional Property of a LiveHTML Page

A LiveHTML page can be declared to be transactional using a directive tag in the page. Using this same tag, the page can also be specified not to support transactions if is called in a transactional execution thread. This tag has the following syntax:

```
<%@ Transaction= attribute %>
```

where *attribute* can be one of the four in Table 6–1, “Transaction attributes and their functions”. Note that this tag can be located anywhere in a LiveHTML page and it is needed to use the transaction objects for Web Application Objects described later in this chapter.

Table 6–1 Transaction attributes and their functions

Attribute	Action	Current Transactional State of Client	LiveHTML Runtime Action
`required`	Specifies that a transaction context is required.	Not transactional	Begin Transaction #2 Execute Scripts End Transaction #2
		In transaction #1	Inherit transaction #1
`requires_new`	Specifies that a new transaction is required. Note that this does not create a nested transaction.	Not transactional	Begin Transaction #2 Execute Scripts End Transaction #2
		In transaction #1	Suspend transaction #1 Begin transaction #2 Execute scripts End transaction #2 Resume transaction #1
`supported`	Specifies that transactions are supported.	Not transactional	Not transactional
		In transaction #1	Inherit transaction #1
`not_supported`	Specifies that transactions are not supported.	Not transactional	Not transactional
		In transaction #1	Suspend transaction #1 Execute scripts Resume transaction #1

As an example, if LiveHTML is currently processing a transaction and a page is called with the tag `<%@ Transaction = not_supported %>` present, LiveHTML will detect the tag, suspend the current transaction, and execute the scripts in the page. When the scripts have been executed, LiveHTML will resume the transaction.

When a “supported” or “requires_new” attribute is specified, LiveHTML will perform the following actions:

1. Check whether the request is associated with an existing transaction.
2. Begins a new transaction if needed (e.g. `<%@ Transaction = requires_new %>`).
3. Commits the transaction upon completion of the page's execution.

Transaction Objects for Web Application Objects

Several objects have been created to enable Web Application Objects (WAO) to participate in transactions. These objects implement methods and attributes and also return constants to help your application participate in transactions. (Note that each LiveHTML page using any of these objects must have the `<%@ Transaction = *attribute* %>` tag specified as described earlier in this chapter.)

Table 6–2 Transaction objects for WAO

Object	Methods	Attributes	Constants

TxContext	commit()
	rollback()
			TX_OUTSIDE
			TX_ROLLBACK
			TX_MIXED
			TX_HAZARD
			TX_PROTOCOL_ERROR
			TX_ERROR
			TX_FAIL
			TX_EINVAL
			TX_COMMITTED
			TX_NO_BEGIN
			TX_ROLLBACK_NO_BEGIN
			TX_MIXED_NO_BEGIN
			TX_HAZARD_NO_BEGIN
			TX_COMMITTED_NO_BEGIN

TxScriptDoc		tx_attr dft_script_language	

Note: The ICXRequest and ICXResponse objects support transactions. They are detailed in Chapter 5, “Developing with Web Application Objects”.

TxContext

This object allows you to commit or rollback a transaction and obtain transactional status information.

Table 6–3 TxContext methods

Method	Description
commit()	Commit current transaction.
rollback()	Rollback current transaction.

Table 6–4 TxContext constants (follows the XA specification by The Open Group)

Attribute	Description
TX_OUTSIDE	value = -1 The transaction is a local transaction.
TX_ROLLBACK	value = -2 The transaction was rolled back.
TX_MIXED	value = -3 The transaction was partially committed and partially rolled back.
TX_HAZARD	value = -4 The transaction may have been partially committed and partially rolled back.
TX_PROTOCOL_ERROR	value = -5 A routine was invoked in an improper context.
TX_ERROR	value = -6 A transient error occurred.
TX_FAIL	value = -7 A fatal error occurred. The request will be allowed to go through but the instance will be destroyed.
TX_EINVAL	value = -8 Invalid arguments were given.
TX_COMMITTED	value = -9 The transaction was heuristically committed.

Table 6–4 TxContext constants (follows the XA specification by The Open Group)

Attribute	Description
TX_NO_BEGIN	value = -100 A transaction was committed and a new transaction could not be started.
TX_ROLLBACK_NO_BEGIN	value = TX_ROLLBACK + TX_NO_BEGIN A transaction was rolled back and a new transaction could not be started.
TX_MIXED_NO_BEGIN	value = TX_MIXED + TX_NO_BEGIN A transaction was partially committed and partially rolled back, and a new transaction could not be started.
TX_HAZARD_NO_BEGIN	value = TX_HAZARD + TX_NO_BEGIN A transaction may have been partially committed and partially rolled back, and a new transaction could not be started.
TX_COMMITTED_NO_BEGIN	value = TX_COMMITTED + TX_NO_BEGIN A transaction was heuristically committed and a new transaction could not be started.

TxScriptDoc

The TxScriptDoc object specifies the transactional attributes of a page.

Table 6–5 TxScriptDoc attributes

Attribute	Read or Read/Write	Type	Description
tx_attr	R	TxAttr	Returns the transaction property of a page. This attribute can have any of the following values: 0 - the page must participate in a transaction 1- the page starts a new transaction 2 - can be executed within a transaction 3 - does not work within a transaction
dft_script_language	R/W	lstring	Specifies the default scripting language used.

Example

The following example illustrates an embedded Perl script in a ".hsp" file requesting the result from a PL/SQL stored procedure using Web Application Objects in a transaction context.

```
...
<%@ Transaction = required %>
...
<%
# Load the package for ICXRequest and create an object reference to ICXRequest
use oracle::OAS::WAO::OASFrmkObject;
my $icx_object = oracle::OAS::WAO::OASFrmkObject->narrow($Server->
                                                get_object("wao://ICXRequest"));

# initialize the object with the URL address of the target
$icx_object->init = ("http://machine/plsqlapp/cartx/procedure_name");

# set the Http request method to GET
$icx_object->set_method(GET);

# create a new ICX connection
$resp_object = $icx_object->connect();

# grab the result of the procedure
$foo = $resp_object->content();
...
%>
```

7

Accessing CORBA Objects from Perl Scripts

IDL (Interface Definition Language) is an industry-standard language used to specify programming interfaces for CORBA objects. CORBA (Common Object Request Broker Architecture) is an industry-standard model for distributed object-oriented programming. Both standards are defined and maintained by the Object Management Group (OMG). To learn about CORBA and IDL, see the OMG web site at **http://www.omg.org**.

When developing a CORBA object, an object developer uses IDL to define and publish the object's public interface. A client programmer then uses any of several available IDL compilers to generate interface bindings for the object in a particular programming language. The client programmer then uses these bindings in that language to access the object from their programs.

Oracle provides an IDL-to-Perl compiler that lets you generate Perl bindings for CORBA objects to make the objects accessible to Perl scripts embedded in your LiveHTML documents. This enhancement allows LiveHTML application developers to take advantage of services exposed by other CORBA objects. It also expands the suite of language and platform choices for LiveHTML developers tremendously.

Contents

Using the IDL-to-Perl Compiler

The IDL-to-Perl compiler is called `perlidlc`, and is located in the following directory on your Oracle Application Server machine:

```
$ORAWEB_HOME/bin
```

You run the IDL-to-Perl compiler as follows:

1. `cd $ORAWEB_HOME/bin`
2. `./perlidlc` *`idl-file-path`*

where *`idl-file-path`* is the full pathname of the IDL file for which you want to generate Perl bindings.

The IDL-to-Perl compiler places its generated files, by default, in two subdirectories of `$ORAWEB_HOME/../cartx/livehtml/stubs/`:

- `perl/`—This subdirectory contains the generated Perl mapped packages (`.pm` files).
- `java/`—This subdirectory contains generated Java mappings required by the Perl mapped packages.

> **Note:** Refer to the <OBJECT> tag section in Chapter 4, "Writing Scripts" for an example of using perlidlc to generate Perl bindings and using these bindings through an object identifier created by the <OBJECT> tag.

Overview of IDL-to-Perl Compiler for Release 4.0

Underlying Java stubs, not Perl stubs, perform runtime range checking of arguments.

Validation of types, in Perl, are currently not being done, so make sure the correct mapped types are given for an IDL operation. Validation of types will be added in another release.

This release supports only static interface invocation and does not support any pseudo object APIs required for dynamic interface invocation.

Other Compiler Options

Other useful options to the IDL-to-Perl compiler include:

- `-I` *`include-dir`*

Specifies a directory, *`include-dir`*, that contains source code files required by the IDL file being compiled. You may repeat this option multiple times to specify multiple directories.

- `-D` *`symbol-name`*

  Specifies a pre-processor symbol, symbol-name, to be considered “defined” during preprocessing (IDL files may contain pre-processor directives in the style of the standard C preprocessor, `cpp`). Specifying this option on the command line is equivalent to using the `#define` preprocessor directive in your source code. You may repeat this option multiple times to define multiple symbols.

- *`-O output-dir`*

  Use the `-O output dir` command to put generated stubs in *`output-dir/`*`perl` and *`output-dir/`*`java.`Ensure that *`output-dir/`*`perl` is in the PERLLIB environment variable of the LiveHTML cartridge and *`output-dir/`*`java` is in the CLASSPATH environment variable of the LiveHTML cartridge, by configuring the **wrb.app** file.

  Java stubs are seamless, which means the Perl programmer is unaware of the Java stub layer.

> **Note:** If *`-O output-dir`* is not specified, `perlidlc` generates stubs, by default, under the `$ORAWEB_HOME/ows/cartx/livehtml/stubs/`*`{perl|java}`* directories. These directories are already included in the PERLLIB and CLASSPATH environments of any LiveHTML cartridge, allowing your LiveHTML pages to seamlessly access stubs generated under default directories. (For NT systems, note that environment variables are limited to 512 bytes each. CLASSPATH may exceed that size.)

Identifiers, Naming Scopes, and Perl Packages

The IDL-to-Perl compiler maps each IDL definition to a Perl identifier of the same name, which can be unscoped or scoped within a package.

The following kinds of IDL definitions create their own naming scopes:

- `module`
- `interface`
- `struct`

- `union`
- `enum`
- `exception`

Note: Perl bindings for unions are not currently implemented.

For example, the compiler would map an unscoped IDL module named `idlmod` to a Perl package named `idlmod` in the newly created file `idlmod.pm` in the `perl/` subdirectory of the output directory. Supposing this module defines a constant `default_width`, a Perl client could then access the constant in this way:

```
use idlmod;
...
$width = $idlmod::default_width;
```

The `use` directive in this example is explained in Accessing Generated Perl Modules below.

Nested Scopes

A scope-creating IDL identifier that is declared within another IDL scope is mapped in the same way as an unscoped identifier, except that the generated Perl module file is stored in a subdirectory named after its parent scope, and the generated Perl package is scoped within its parent package.

For example, for an IDL module `outer` that declares a submodule `inner`, the IDL-to-Perl compiler would generate at least these two Perl module files in the `perl/` subdirectory of the output directory:

- `outer.pm`
- `outer/inner.pm`

Supposing the module `inner` declares a constant `right`, a Perl client could then access the constant this way:

```
use outer;
...
$align = $outer::inner::right;
```

Accessing Generated Perl Modules

To access a mapped package, you must give the package name as the argument of a `use` or `require` directive. For example:

```
use idlmod;
```

You must specify such a `use` or `require` directive for each mapped package you want to access.

When you explicitly include a mapped package for an IDL identifier `<ident>` in your script, mapped packages for all IDL identifiers scoped within `<ident>` are automatically included. For example, the `use outer;` command in the above example includes `outer` and `outer::inner`, which means accessing `$outer::inner::right;` is allowable.

Data Types

Perl does not differentiate between various scalar types. The IDL-to-Perl compiler maps most basic IDL data types to the Perl scalar type. To compensate for the resulting lack of compile-time type checking in your Perl script, the generated Perl code for calls to CORBA object operations perform runtime range validation on values passed for parameters mapped to scalar type.

For parameters mapped to other Perl types, such as array reference and hash reference, the generated Perl code checks whether the passed values are of the appropriate Perl mapped type.

For constructed types, such as IDL structures and unions (described in detail below in Using the Generated Perl Bindings), the generated Perl code recursively performs type checking and range validation on the fields of the constructed types.

Using the Generated Perl Bindings

This section illustrates how Perl clients of CORBA objects can use the bindings generated by the IDL-to-Perl compiler. Each sub-section below presents an example of a particular kind of IDL definition along with a corresponding example of Perl code illustrating client usage.

Modules

An IDL module is a scope-creating definition. The IDL-to-Perl compiler maps an IDL module to a Perl package that provides the scope for the identifiers declared in the module. Each module name must be unique within the local ORB system.

Example

```
// IDL
module finance {
    const long L = 3;

    ...
};
```

Perl Client Usage

```
# Perl client
use finance;
...
$longval = $finance::L;
```

Example: Nested Modules

The IDL-to-Perl compiler maps an IDL module declared within another IDL module to a package contained within the package representing the parent module:

```
// IDL
module outer {
    module inner {
        const short thing = 3;
        ...
    };
    ...
};
```

Perl Client Usage

```
# Perl client
use outer;
...
$intval = $outer::inner::thing;
# outer::inner package is automatically included while including outer
```

Object References

The interface to a CORBA object is defined in IDL as an `interface` definition. To invoke operations on a CORBA object from your Perl client program, you must obtain a reference to the object. To allow this, the IDL-to-Perl compiler makes available a `bind()` class (package) method in every package that represents the interface to a CORBA object. `bind()` creates and returns a reference to a proxy object of the corresponding CORBA server object.

Invoke this, and all other class methods without reference to an object, by qualifying the method name with the package name.

Example

```
// IDL
module finance {
    interface account {
        ...
    }

    interface bank {
        account newaccount(in string name);
        account getaccount(in string name);
        ...
    }
    ...
}
```

Perl Client Usage

To get a reference to an object of type `finance::bank`, the Perl client code must use the `bind()` method that `interface bank` inherits from `CORBA::Object`:

```
# Perl client
use finance;
...
```

Note: You must use the arrow operator (`->`) to call these methods rather than the package delimiter (`::`) operator. The Perl5 interpreter requires the use of the arrow operator to process arguments for these methods correctly.

You can then use the object reference `$bank` to invoke methods on the account object:

```
# Perl client
...
$account = $bank->getaccount("Joseph P. Shmuck");
```

Note: The client program's first call to the `bind()` method of any package automatically initializes the ORB for use by subsequent object and ORB calls from the client.

Narrowing

Suppose you have a reference to an object implementing interface A, and you know this reference actually refers to an object implementing interface B. You can "narrow" the reference by obtaining a reference to an object implementing interface B. To do this, use the narrow() class method of class B that maps interface B, such as:

```
$Bref = B->narrow($Aref);
```

If you are mistaken, and $Aref does not really refer to an object that implements interface B, this method raises an exception (see Exceptions below).

Interfaces

The IDL-to-Perl compiler maps an IDL interface to a Perl package.

Example

```
// IDL
module finance {
    interface bank {
        const short stuff = 3;
        ...
    };
    ...
};
```

The IDL interface bank is mapped to the finance::bank package.

Perl Client Usage

```
# Perl client
use finance; #also uses finance::bank
...
$var = $finance::bank::stuff;
```

Example: Inheritance

Mapping of the IDL inheritance is achieved through Perl's inheritance mechanism on the mapped packages of the IDL interfaces.

```
// IDL
module finance {
    interface account {
        double getbalance();
        ...;
    };
```

```
        interface checkingaccount : account {
            ...
        };
        interface bank {
            checkingaccount newchecking(in string name);
            checkingaccount getchecking(in string name);
            ...
        }
    };
```

Perl Client Usage

```
    # Perl client
    use finance;
    ...
    $bank = finance::bank->bind();
    $checking = $bank->getchecking("Joseph P. Shmuck");
    $balance = $checking->getbalance();
```

Perl mapped package `finance::checkingaccount` inherits from `finance::account`, so invoking the `getbalance` method on `$checking` is valid because `$checking`, an instance of `finance::checkingaccount`, inherits `getbalance` from `finance::account`.

Constants

The IDL-to-Perl compiler maps an IDL constant to a Perl scalar in the package corresponding to the IDL scope in which the constant is declared.

Example

```
    // IDL
    module finance {
        interface account {
            const double minbalance = 500.0;
            ...
        };
        ...
    };
```

Perl Client Usage

```
    # Perl client
    use finance;
    ...
    $minbal = $finance::account::minbalance;
```

Basic Data Types

The IDL-to-Perl compiler maps the following IDL types to the Perl scalar type:

- short
- long
- long long
- unsigned short
- unsigned long
- unsigned long long
- float
- double
- char
- wchar
- octet
- boolean

See Data Types above for a discussion of how the generated Perl code performs runtime type and range checking.

Values for variables, parameters, and operation return values of IDL type boolean map to Perl scalar values, which can be evaluated in a boolean context.

The IDL Any Type

The IDL-to-Perl compiler maps a variable or parameter of type any to a reference to a CORBA pseudo-object of type CORBA::Any. See Chapter 8, “CORBA Pseudo-Object API for Perl Clients” to learn about this pseudo-object interface.

Example

```
// IDL
module finance {
    interface account {
        Any collateral(in any asset);
        ...
    };
    interface bank {
        account newaccount(in string name);
        account getaccount(in string name);
        ...
    };
};
```

Perl Client Usage

```
# Perl Client
use finance;
...
$bank = finance::bank->bind();
$acct = $bank->getaccount("Joseph P. Shmuck");

# create a CORBA::Any pseudo-object and get a reference to it
$asset = CORBA::TypeCode->create_any();

# for some value, construct a TypeCode and store it in $tc--
# suppose for this example it's a short integer type
$value = 12;
$tc = CORBA::TypeCode->get_primitive_tc($CORBA::TCKind::tk_short);

# initialize the CORBA::Any pseudo-object with this value and typecode
$asset->insert($value, $tc);

# pass the CORBA::Any reference as a parameter
$col = $acct->collateral($asset);

# do something with the returned CORBA::Any reference
$ctc = $col->type();
if ($ctc->kind() == $CORBA::TCKind::tc_string) {
    # get string value from the CORBA::Any pseudo-object referred to by $col
    $val = $col->extract();
    ...
}
...
```

Strings

The IDL-to-Perl compiler maps both bounded and unbounded IDL strings to Perl scalar. Because Perl scalar strings are always unbounded, the generated Perl code for method invocations truncates string values passed for bounded string method parameters if the values exceed the allowed length.

Example

```
// IDL
module finance {
    interface account {
        double totaldeposits(in string frombankid, in string<8> date);
        ...
    };
    interface bank {
        account newaccount(in string name);
        account getaccount(in string name);
        ...
    };
    ...
};
```

Perl Client Usage

```
# Perl client
use finance;
...
$bank = finance::bank->bind();
$acct = $bank->getaccount("Joseph P. Shmuck");

# the date string passed below will be truncated to 8 characters,
# lopping off " extra chars"
$totaldeposits = $acct->totaldeposits("90-7005", "19980106 extra chars");
...
```

Arrays and Sequences

The IDL-to-Perl compiler maps IDL arrays and IDL sequences to Perl array references.

Example

```
// IDL
module finance {
    interface account {
        ...
    };
    interface bank {
        long getaccounts(out sequence<account> accounts);
        account newjointaccount(in sequence<string, 2> names);
        ...
    };
```

```
        ...
    };
```

Perl Client Usage

```
    # Perl client
    use finance;
    ...
    $bank = finance::bank->bind();

    # pass reference to the array which getaccounts is to populate
    $numaccts = $bank->getaccounts($accts);
    for ($i = 0; $i < $numaccts; $i++) {
        $account = $accts->[$i];
        ...
    }

    # construct and pass an array reference to newjointaccount
    $names = [ "Joseph P. Shmuck", "Josephine Q. Public" ];
    $jointacct = $bank->newjointaccount($names);
    ...
```

Operations

The IDL-to-Perl compiler maps an IDL operation declared within an interface to a Perl method in the package that represents the interface. See Interfaces above for an example.

Attributes

The IDL-to-Perl compiler maps an IDL attribute declared within an interface to an overloaded Perl method named after the attribute in the package that represents the interface. You can call the method in two ways:

- When called with no arguments, the method returns the current value of the attribute.
- When called with one argument, the method sets the value of the attribute to the value of the argument.

Example

```
    // IDL
    module finance{
        interface account {
            attribute double interestrate;
            ...
        };
        interface bank {
            account newaccount(in string name);
            account getaccount(in string name);
            ...
        }
        ...
    };
```

Perl Client Usage

```
    # Perl client
    use finance;
    ...
    $bank = finance::bank->bind();
    $acct = $bank->getaccount("Joseph P. Shmuck");
    $oldrate = $acct->interestrate();
    $acct->interestrate($oldrate + 0.005);
```

Enumerated Types

For an enumerated type, the IDL-to-Perl compiler generates a package that declares scalar variables named after the enumerators of the type. These scalar variables are assigned increasing integer values in order starting with zero. The package name is the same as the IDL name for the enumerated type, hence, the enumerated type maps to a Perl package. A variable of the enumerated type maps to a Perl scalar.

Example

```
    // IDL
    module finance {
        interface account {
            enum trans_type { deposit, withdrawl };
            attribute trans_type lasttranstype;
            ...
        };
        interface bank {
            account newaccount(in string name);
            account getaccount(in string name);
            ...
        }
        ...
    };
```

Perl Client Usage

```
# Perl client
use finance;
...
$bank = finance::bank->bind();
$acct = $bank->getacount("Joseph P. Shmuck");
$lasttranstype = $acct->lasttranstype();

# test value against an enumerator
if ($lasttranstype == $finance::account::trans_type::deposit) {
    ...
}
```

Structures

For an IDL structure, the IDL-to-Perl compiler automatically generates a Perl package named after the structure. A variable of this structure type is mapped to a reference to a hash. The hash contains keys named after the structure elements.

Example

```
// IDL
module finance {
    interface account
        enum trans_type { deposit, withdrawl };
        struct transaction {
            string<8> date;
            trans_type type;
            double amount;
        };
        short do_transaction(inout transaction trans);
        ...
    };
    interface bank {
        account newaccount(in string name);
        account getaccount(in string name);
        ...
    }
    ...
};
```

Perl Client Usage

```
# Perl client
use finance;
```

```
...
$bank = finance::bank->bind();
$acct = $bank->getaccount("Jospeh P. Shmuck");

# construct a hash reperesenting a variable of IDL type transaction
$trans = {};
$trans->{date} = "19980106";
$trans->{type} = $finance::account::trans_type::withdrawl;
$trans->{amount} = 40.0;

# pass a reference to this hash as a parameter to an operation
$status = $acct->do_transaction($trans);
...
```

Unions

Note: Perl bindings for unions are not currently implemented.

Typedefs

An IDL typedef defines a new type that is an alias for another IDL type, which is a typedef or a base IDL type. An IDL typedef maps to a Perl package that represents this new type. This kind of mapping is used mostly by the runtime system for marshalling/unmarshalling arguments. A typedef variable maps according to the rules for the unwound base type for which it has an alias.

Example

```
// IDL
module finance {
    interface account {
        typedef enum trans_type { deposit, withdrawl } trans_t;
        attribute trans_t lasttranstype;
        ...
    };
    interface bank {
        account newaccount(in string name);
        account getaccount(in string name);
        ...
    }
    ...
};
```

Perl Client Usage

```
# Perl Client
use finance;
...
$bank = finance::bank->bind();
$lastttranstype = $acct->lasttranstype();

# test value against an enumerator--
# must specify the base type here rather than the defined type
if ($lasttranstype == $finance::account::trans_type::deposit) {
    ...
}
...
```

Exceptions

Note: The implementation described here will be replaced in a future release by a true exception mechanism.

IDL exceptions are mapped very similarly to IDL structures. For an IDL exception, the IDL-to-Perl compiler generates a Perl package named after the exception. A variable of this exception type is mapped to a reference to a hash. The hash contains keys named after the exception elements.

An IDL operation raises an exception by returning a reference to the exception in place of its usual return value. The correct practice when calling such an operation from Perl is to use a class method of the class `Oracle::hlpr::Excp` to test whether the returned value is an exception. If it is, you can use another class method of the `Oracle::hlpr::Excp` class to rethrow the exception.

Currently, there is not a way to see if an exception is of a particular type, or to print the exception.

This release does not offer enhanced exception handling support. An exception package will be provided in a future release.

Unmarshalling of user defined exceptions thrown by CORBA server objects is absent in this release. The user cannot access the fields of the user defined exception. Support for this will be added in a future release.

CORBA system exceptions need to be mapped to Perl and will be done in a future release.

`Oracle::hlpr::Excp` provides the following class methods:

- `isexcp()`—Called with a single argument, this method returns a true scalar value if the argument is a reference to an exception object.
- `throw()`—Called with a single argument, this method throws the exception referred to by the argument; the argument must be a reference to a exception object.
- `rethrow()`—Called with no arguments, this method rethrows the last exception.

Example

```
// IDL
module finance {
    interface account {
        typedef enum trans_type { deposit, withdrawl } trans_t;
        attribute trans_t lasttranstype;
        ...
    };
    interface bank {
        exception badaccount {};
        account newaccount(in string name);
        account getaccount(in string name) raises(badaccount);
        ...
    }
    ...
};
```

Perl Client Usage

```
# Perl Client
use finance;
use Oracle::hlpr::Excp;
...
$bank = finance::bank->bind();

$acct = $bank->getaccount("Joseph P. Shmuck");
if (Oracle::hlpr::Excp->isexcp($acct)) {
    Oracle::hlpr::Excp->rethrow();
}
```

8

CORBA Pseudo-Object API for Perl Clients

Oracle Application Server provides Perl bindings for the CORBA pseudo-object interfaces required to support the Static Invocation Interface (SII) from Perl clients. This chapter provides sample Perl code illustrating calls to the supported pseudo-object interfaces. This chapter does not discuss the calling semantics for most of these interfaces, which is defined in *The Common Object Request Broker: Architecture and Specification*, available at the Object Management Group web site, **http://www.omg.org**. Some of the interfaces, however, are specific to the Perl implementation of the pseudo-objects; in these cases, the calling semantics are described.

> **Note:** Oracle Application Server does not provide complete Perl bindings for these pseudo-objects; each section below identifies the subset of each pseudo-object interface for which bindings are provided, and any special interfaces defined by the Perl bindings themselves.

Contents

This chapter documents the Perl bindings for the following CORBA pseudo-object interfaces:

- Object
- ORB
- Any
- TypeCode
- TCKind

Object

Perl bindings are provided for the following subset of the `Object` interface:

```
// PIDL
module CORBA {
    interface Object {
        // instance methods
        Object duplicate();
        void release();
        boolean is_a(in string logical_type_id);
        boolean non_existent();
        boolean is_equivalent(in Object other_object);
    };
};
```

The Perl bindings for these operations use the same names prefixed by underscore (‘_’) characters. For example, the `duplicate()` operation is mapped to `_duplicate()` in Perl.

Instance Methods

In all the following examples, *`$obj`* is a reference to a CORBA object.

duplicate()

```
// PIDL
Object duplicate();
```

Sample call from Perl

```
# Perl client
use CORBA::Object;
...
$dup = $obj->_duplicate();
```

release()

```
// PIDL
void release();
```

Sample call from Perl

```
# Perl client
use CORBA::Object;
...
$obj->_release();
```

is_a()

```
// PIDL
boolean is_a(in string logical_type_id);
```

Sample call from Perl

In this example, *$logical_id* is of Perl scalar type with a string value.

```
# Perl client
use CORBA::Object;
...
if ($obj->_is_a($logical_id)) {
    ...
}
```

non_existent()

```
// PIDL
boolean non_existent();
```

Sample call from Perl

```
# Perl client
use CORBA::Object;
...
if ($obj->_non_existent()) {
    ...
}
```

is_equivalent()

```
// PIDL
boolean is_equivalent(in Object other_object);
```

Sample call from Perl

In this example, *$other_object* is a reference to a CORBA object.

```
# Perl client
use CORBA::Object;
...
if ($obj->_is_equivalent($other_object)) {
    ...
}
```

ORB

Perl bindings are provided for the following subset of the ORB interface:

```
// PIDL
module CORBA {
    interface ORB {
        typedef string ObjectId;
        typedef sequence<ObjectId> ObjectIdList;

        exception InvalidName{};

        // class methods
        ObjectIdList list_initial_services();
        Object resolve_initial_references(in ObjectID identifier)
            raises(InvalidName);

        string object_to_string(in Object obj);
        Object string_to_object(in string str);

        // interfaces defined by the Perl bindings--
        // use the init() operations in place of CORBA::ORB_init()
        ORB init();
        ORB init(inout sequence<string> argv);

        // instance methods
        Object bind(string object_id);
        void term();
    };
};
```

Class Methods

list_initial_services()

```
// PIDL
ObjectIdList list_initial_services();
```

Sample call from Perl

```
# Perl client
use CORBA::ORB;
...
$servlist = CORBA::ORB->list_initial_services();
$numservs = scalar(@$servlist);
for ($i = 0; $i < $numservs; $i++) {
    $service = $serlist->[$i];
    ...
}
```

resolve_initial_references()

```
    // PIDL
    Object resolve_initial_references(in ObjectID identifier)
        raises(InvalidName);
```

Sample call from Perl

In this example, *$obj_id* is of scalar type with a string value representing a value of IDL type ObjectID. This value should be one of the values returned by list_initial_services().

```
    # Perl client
    Fuse Oracle::hlpr::Excp;
    ...
    $obj = CORBA::ORB->resolve_initial_references($obj_id);
    if (Oracle::hlpr::Excp->isexcp($obj)) {
        Oracle::hlpr::Excp->throw($obj);
    }
```

object_to_string()

```
    // PIDL
    string object_to_string(in Object obj);
```

Sample call from Perl

In this example, *$obj* is a reference to a CORBA object.

```
    # Perl client
    use CORBA::ORB;
    ...
    $str = CORBA::ORB->object_to_string($obj);
```

string_to_object()

```
    // PIDL
    Object string_to_object(in string str);
```

Sample call from Perl

In this example, *$string* is of scalar type with a string value.

```
# Perl client
use CORBA::ORB;
...
$obj = CORBA::ORB->string_to_object($string);
```

init()

Caution: Do not call this method from within a LiveHTML document. When your script runs from within a LiveHTML document, your script automatically has access to a running ORB.

You use this operation to initialize the Object Request Broker (ORB) and invoke services on it.

```
// PIDL
ORB init();
ORB init(inout sequence<string> argv);
```

When using the second form of `init()`, the first element of the `argv` array must be a valid Java CLASSPATH value, which must specify the location or locations of the Java bindings for CORBA pseudo-objects; you must make sure the directory `$ORAWEB_HOME/../cartx/livehtml/stubs/java/` is included in the CLASSPATH. This is required because the IDL-to-Perl compiler generates Java bindings that the Perl bindings use to invoke operations on the ORB and CORBA objects.

Sample call from Perl

```
# Perl client
use CORBA::ORB;
...
$orb = CORBA::ORB->init();
```

Instance Methods

bind()

The method binds to the CORBA object identified by its argument.

```
// PIDL
Object bind(string object_id);
```

Sample call from Perl

In this example:

- *$orb* is a reference to a CORBA ORB pseudo-object.
- *$object_id* is of scalar type with a string value.

```
# Perl client
use CORBA::ORB;
...
$obj = $orb->bind($object_id)
```

term()

Caution: Do not call this method from within a LiveHTML document. When your script runs from within a LiveHTML document, you must not terminate the ORB that the LiveHTML cartridge is using.

You use this method to terminate the ORB. When this method returns, no further invocations on CORBA objects are possible.

```
// PIDL
void term();
```

Sample call from Perl

In this example, *$orb* is a reference to a CORBA ORB pseudo-object.

```
# Perl client
use CORBA::ORB;
...
$orb->term();
```

Any

Perl bindings are provided for the following subset of the Any interface:

```
// PIDL
module CORBA {
    interface Any {
        // instance methods
        TypeCode type();
        void insert(in any value, in TypeCode type);
        any extract();    // actually returns a value of type
                          // set by insert()
    };
};
```

Instance Methods

In these examples, *$any* is a reference to a CORBA Any pseudo-object.

type()

```
// PIDL
TypeCode type();
```

Sample call from Perl

```
# Perl client
use CORBA::Any;
...
$type = $any->type();
```

insert()

In this PIDL, *anyval* represents the type encoded in the `type` parameter.

```
// PIDL
void insert(in anyval value, in TypeCode type);
```

Sample call from Perl

In this example:

- *$value* is a value or a reference to a value of the type indicated by *$type*.
- *$type* is a reference to a CORBA `TypeCode` pseudo-object.

> **Caution:** This value must not refer to a typecode that was constructed using the `create_*_tc()` methods of `CORBA::TypeCode`. You may use the typecode returned by the `_type()` method of a CORBA object, or the typecode for a primitive type returned by the `get_primitive_tc()` method of `CORBA::TypeCode`.

```
# Perl client
use CORBA::Any;
use CORBA::TypeCode;
...

$any->insert($value, $type);
```

extract()

In this PIDL, *anyval* represents the type that was specified by the `insert()` call that initialized the pseudo-object.

```
// PIDL
anyval extract();    // actually returns a value of type set by insert()
```

Sample call from Perl

```
# Perl client
use CORBA::Any;
use CORBA::TypeCode;
use CORBA::TCKind;
...
$type = $any->type();
$val = $any->extract();
$kind = $type->kind();
if ($kind == $CORBA::TCKind::tk_short) {
    ...
}
elsif ($kind == $CORBA::TCKind::tk_long) {
    ...
}
```

Generally, in situations similar to this example, the context indicates what `$kind` is likely to be; it is not usually necessary to test against every possible `TCKind` value.

TypeCode

The CORBA specification declares the class methods listed here in `interface ORB`. The Perl bindings, however, implement these methods as class methods of `interface TypeCode` as shown below:

```
// PIDL
module CORBA {
    struct StructMember {
        string name;
        TypeCode type;
        IDLType type_def; // currently not used
    }
    typedef sequence<StructMember> StructMemberSeq;

    typedef sequence<string> EnumMemberSeq;

    interface TypeCode {
```

```
// class methods
Any create_any();
TypeCode create_struct_tc (
    in string repository_id,
    in string type_name,
    in StructMemberSeq members
);
TypeCode create_enum_tc (
    in string repository_id,
    in string type_name,
    EnumMemberSeq members
);
TypeCode create_alias_tc (
    in string repository_id,
    in string type_name,
    in TypeCode original_type
);
TypeCode create_exception_tc (
    in string repository_id,
    in string type_name,
    in StructMemberSeq members
);
TypeCode create_interface_tc (
    in string repository_id,
    in string type_name
);
TypeCode create_string_tc (
    in unsigned long bound
);
TypeCode create_wstring_tc (
    in unsigned long bound
);
TypeCode create_sequence_tc (
    in unsigned long bound,
    in TypeCode type
);
TypeCode create_array_tc (
    in unsigned long length,
    in TypeCode type
);

// class method defined by the Perl bindings
TypeCode get_primitive_tc(TCKind kind);

// exceptions raised by instance methods
```

```
            exception Bounds {};
            exception BadKind {};

            // instance methods
            boolean equal(in TypeCode tc);
            TCKind kind();
            string id() raises(BadKind);
            string name() raises(BadKind);
            unsigned long member_count() raises(BadKind);
            string member_name(
                in unsigned long index
            ) raises(BadKind, Bounds);
            TypeCode member_type(
                in unsigned long index
            ) raises(BadKind, Bounds);
            any member_label(
                in unsigned long index
            ) raises(BadKind, Bounds);
            TypeCode discriminator_type() raises(BadKind);
            long default_index() raises(BadKind);
            unsigned long length() raises(BadKind);
            TypeCode content_type() raises(BadKind);

            // instance method defined by the Perl bindings
            TypeCode orig_type();
        };
    };
```

Class Methods

The CORBA specification declares these methods in `interface ORB`. The Perl bindings, however, implement these methods as class methods of `interface TypeCode`.

In the following examples, *`$repository_id`* `and` *`$type_name`* are each of scalar type with a string value.

create_any()

```
// PIDL
Any create_any();
```

Sample Call from Perl

```
# Perl client
use CORBA::TypeCode;
```

```
...
$any = CORBA::TypeCode->create_any();
```

create_struct_tc()

```
// PIDL
TypeCode create_struct_tc (
    in string repository_id,
    in string type_name,
    in StructMemberSeq members
);
```

Sample call from Perl

In this example:

- *`$member_name1`* and *`$member_name2`* are of scalar type with a string value.
- *`$member_type1`* and *`$member_type2`* are references to CORBA `TypeCode` pseudo-objects.

```
# Perl client
use CORBA::TypeCode;
...
# construct references to hashes defining structure members
$member1 =
  { "name" => $member_name1, "type" => $member_type1, "type_def" => "" };
$member2 =
  { "name" => $member_name2, "type" => $member_type2, "type_def" => "" };
# repeat for additional structure members

# construct reference to array of members, which defines the structure
$members = [ $member1, $member2 ];
$newtypecode =
  CORBA::TypeCode->create_struct_tc($repository_id, $type_name, $members);
```

create_enum_tc()

```
// PIDL
TypeCode create_enum_tc (
    in string repository_id,
    in string type_name,
    EnumMemberSeq members
);
```

Sample call from Perl

```
# Perl client
use CORBA::TypeCode;
...
# construct a reference to an array containing the new enum value names
$enumvals = [ "val1", "val2", "val3" ];
$newtypecode =
  CORBA::TypeCode->create_enum_tc($repository_id, $type_name, $enumvals);
```

create_alias_tc()

```
// PIDL
TypeCode create_alias_tc (
    in string repository_id,
    in string type_name,
    in TypeCode original_type
);
```

Sample call from Perl

In this example, *$original_type* is a reference to a CORBA TypeCode pseudo-object.

```
# Perl client
use CORBA::TypeCode;
...
$newtypecode =
  CORBA::TypeCode->create_alias_tc($repository_id, $type_name,
  $original_type);
```

create_exception_tc()

```
// PIDL
TypeCode create_exception_tc (
    in string repository_id,
    in string type_name,
    in StructMemberSeq members
);
```

Sample call from Perl

```
# Perl client
use CORBA::TypeCode;
...
```

```
# construct references to hashes defining exception members
$member1 =
  { "name" => $member_name1, "type" => $member_type1, "type_def" => "" };
$member2 =
  { "name" => $member_name2, "type" => $member_type2, "type_def" => "" };
# repeat for additional exception members

# construct reference to array of members, which defines the exception
$members = [ $member1, $member2 ];
$newtypecode =
  CORBA::TypeCode->create_struct_tc($repository_id, $type_name, $members);
```

create_interface_tc()

```
// PIDL
TypeCode create_interface_tc (
    in string repository_id,
    in string type_name
);
```

Sample call from Perl

```
# Perl client
use CORBA::TypeCode;
...
$newtypecode = CORBA::TypeCode->($repository_id, $type_name);
```

create_string_tc()

```
// PIDL
TypeCode create_string_tc (
    in unsigned long bound
);
```

Sample call from Perl

In this example, *$bound* is of scalar type with an unsigned long integer value.

```
# Perl client
use CORBA::TypeCode;
...
$newtypecode = CORBA::TypeCode->create_string_tc($bound);
```

create_wstring_tc()

```
// PIDL
TypeCode create_wstring_tc (
    in unsigned long bound (
);
```

Sample call from Perl

In this example, *$bound* is of scalar type with an unsigned long integer value.

```
    # Perl client
    use CORBA::TypeCode;
    ...
    $newtypecode = CORBA::TypeCode->create_wstring_tc($bound);
```

create_sequence_tc()

```
    // PIDL
    TypeCode create_sequence_tc (
        in unsigned long bound,
        in TypeCode type
    );
```

Sample call from Perl

In this example:

- *$bound* is of scalar type with an unsigned long integer value.
- *$type* is a reference to a CORBA TypeCode pseudo-object.

```
    # Perl client
    use CORBA::TypeCode;
    ...
    $newtypecode = CORBA::create_sequence_tc($bound, $type);
```

create_array_tc()

```
    // PIDL
    TypeCode create_array_tc (
        in unsigned long length,
        in TypeCode type
    );
```

Sample call from Perl

In this example:

- *$length* is of scalar type with an unsigned long integer value.
- *$type* is a reference to a CORBA TypeCode pseudo-object.

```
# Perl client
use CORBA::TypeCode;
...
$newtypecode = CORBA::create_array_tc($length, $type);
```

get_primitive_tc()

```
// PIDL
TypeCode get_primitive_tc(TCKind kind);
```

Sample call from Perl

In this example, *$kind* is of scalar type with a value defined by `enum TCKind`; see TCKind below.

```
# Perl client
use CORBA::TypeCode;
use CORBA::TCKind;
...
$typecode = CORBA::get_primitive_tc($kind);
```

Instance Methods

In these examples, *$typecode* is a reference to a CORBA `TypeCode` pseudo-object.

equal()

```
// PIDL
boolean equal(in TypeCode tc);
```

Sample call from Perl

In this example, *$other_typecode* is a reference to a CORBA TypeCode pseudo-object.

```
# Perl client
use CORBA::TypeCode;
...
if ($typecode->equal($other_typecode)) {
    ...
}
```

kind()

```
// PIDL
TCKind kind();
```

Sample call from Perl

```
# Perl client
use CORBA::TypeCode;
...
$kind = $typecode->kind();
if ($kind == $CORBA::TCKind::tk_short) {
    ...
}
```

id()

```
// PIDL
string id() raises(BadKind);
```

Sample call from Perl

```
# Perl client
use CORBA::TypeCode;
use Oracle::hlpr::Excp;
...
$id = $typecode->id();
if (Oracle::hlpr::Excp->isexcp($id)) {
    Oracle::hlpr::Excp->throw($id);
}
```

name()

```
// PIDL
string name() raises(BadKind);
```

Sample call from Perl

```
# Perl client
use CORBA::TypeCode;
use Oracle::hlpr::Excp;
...
$name = $typecode->name();
if (Oracle::hlpr::Excp->isexcp($name)) {
    Oracle::hlpr::Excp->throw($name);
}
```

member_count()

```
// PIDL
unsigned long member_count() raises(BadKind);
```

Sample call from Perl

```
# Perl client
use CORBA::TypeCode;
use Oracle::hlpr::Excp;
...
$num_members = $typecode->member_count();
if (Oracle::hlpr::Excp->isexcp($num_members)) {
    Oracle::hlpr::Excp->throw($num_members);
}
```

member_name()

```
// PIDL
string member_name(
    in unsigned long index
) raises(BadKind, Bounds);
```

Sample call from Perl

In this example, *$index* is of scalar value with an unsigned long integer value.

```
# Perl client
use CORBA::TypeCode;
use Oracle::hlpr::Excp;
...
$mem_name = $typecode->member_name($index);
if (Oracle::hlpr::Excp->isexcp($mem_name)) {
    Oracle::hlpr::Excp->throw($mem_name);
}
```

member_type()

```
// PIDL
TypeCode member_type(
    in unsigned long index
) raises(BadKind, Bounds);
```

Sample call from Perl

In this example, *$index* is of scalar value with an unsigned long integer value.

```
# Perl client
use CORBA::TypeCode;
use Oracle::hlpr::Excp;
...
$mem_type = $typecode->member_type($index);
```

```
if (Oracle::hlpr::Excp->isexcp($mem_type)) {
    Oracle::hlpr::Excp->throw($mem_type);
}
```

member_label()

```
// PIDL
any member_label(
    in unsigned long index
) raises(BadKind, Bounds);
```

Sample call from Perl

In this example, *$index* is of scalar value with an unsigned long integer value.

```
# Perl client
use CORBA::TypeCode;
use Oracle::hlpr::Excp;
...
$label = $typecode->member_label($index);
if (Oracle::hlpr::Excp->isexcp($label)) {
    Oracle::hlpr::Excp->throw($label);
}
```

discriminator_type()

```
// PIDL
TypeCode discriminator_type() raises(BadKind);
```

Sample call from Perl

```
# Perl client
use CORBA::TypeCode;
use Oracle::hlpr::Excp;
...
$tc = $typecode->discriminator_type();
if (Oracle::hlpr::Excp->isexcp($tc)) {
    Oracle::hlpr::Excp->throw($tc);
}
```

default_index()

```
// PIDL
long default_index() raises(BadKind);
```

Sample call from Perl

```
# Perl client
use CORBA::TypeCode;
use Oracle::hlpr::Excp;
...
$def_index = $typecode->default_index();
if (Oracle::hlpr::Excp->isexcp($def_index)) {
    Oracle::hlpr::Excp->throw($def_index);
}
```

length()

```
// PIDL
unsigned long length() raises(BadKind);
```

Sample call from Perl

```
# Perl client
use CORBA::TypeCode;
use Oracle::hlpr::Excp;
...
$len = $typecode->length();
if (Oracle::hlpr::Excp->isexcp($len)) {
    Oracle::hlpr::Excp->throw($len);
}
```

content_type()

```
// PIDL
TypeCode content_type() raises(BadKind);
```

Sample call from Perl

```
# Perl client
use CORBA::TypeCode;
use Oracle::hlpr::Excp;
...
$tc = $typecode->content_type();
if (Oracle::hlpr::Excp->isexcp($tc)) {
    Oracle::hlpr::Excp->throw($tc);
}
```

orig_type()

```
// PIDL
TypeCode orig_type();
```

Sample call from Perl

```
# Perl client
use CORBA::TypeCode;
...
$tc = $typecode->orig_type();
```

TCKind

Perl bindings are provided for `enum TCKind`:

```
// PIDL
module CORBA {
    enum TCKind {
        tk_null, tk_void,
        tk_short, tk_long, tk_ushort, tk_ulong,
        tk_float, tk_double, tk_boolean, tk_char,
        tk_octet, tk_any, tk_TypeCode, tk_Principal, tk_objref,
        tk_struct, tk_union, tk_enum, tk_string,
        tk_sequence, tk_array, tk_alias, tk_except,
        tk_longlong, tk_ulonglong, tk_longdouble,
        tl_wchar, tk_wstring, tk_fixed, tk_byte
    };
};
```

The Perl bindings define the `tk_byte` value; it is not declared in the CORBA specification.

For an example of using TCKind, see the kind() method of the TypeCode pseudo-object, above.

9

Sample Output from the IDL-to-Perl Compiler

This chapter details the directory structure and Perl module files generated by the IDL-to-Perl compiler in compiling the following sample IDL.

Contents

Sample IDL

This IDL summarizes the examples used in Chapter 7, “Accessing CORBA Objects from Perl Scripts”:

```
// IDL

module finance {
    const long L = 3;

    interface account {
        const double minbalance = 500.0;
        typedef enum trans_type { deposit, withdrawl } trans_t;

        struct transaction {
            string<8> date;
            trans_t type;
            double amount;
        };
        attribute double interestrate;
        attribute trans_t lasttranstype;

        double getbalance();
        double totaldeposits(in string frombankid, in string<8> date);
        short do_transaction(inout transaction trans);

        any collateral(in any asset);
    };

    interface checkingaccount : account {

    };

    interface bank {
        const short stuff = 3;
        typedef sequence<string, 2> stringseq;
        typedef sequence<account> accountseq;

        account newaccount(in string name);
        account newjointaccount(in stringseq names);
        checkingaccount newchecking(in string name);

        exception badaccount {};

        account getaccount(in string name) raises(badaccount);
        long getaccounts(out accountseq accounts) raises(badaccount);
        checkingaccount getchecking(in string name) raises(badaccount);
    };
};

module outer {
    module inner {
        const short thing = 3;
    };
};
```

Directory Structure of the Generated Files

The following command line was used to compile the above IDL, contained in the file `ex.idl`:

```
perlidlc ex.idl
```

This command created the following files and directories in the directory `$ORAWEB_HOME/../cartx/livehtml/stubs/perl/`. Directories are indicated by trailing Unix-style slash ‘/’ characters:

```
finance/
    account/
        trans_t.pm
        trans_type.pm
        transaction.pm
    account.pm
    bank/
        accountseq.pm
        badaccount.pm
        stringseq.pm
    bank.pm
    checkingaccount.pm
finance.pm
outer/
    inner.pm
outer.pm
```

Listings of the Generated Files

The following files are listed in the order they appear in the previous section.

finance/account/trans_t.pm

```
package finance::account::trans_t;

use Oracle::Java::VM qw(:TYPES);
use Oracle::Java::ClassLdr;
use Oracle::Java::Object;
use Oracle::hlpr::Excp;
use CORBA::TypeCode;
use Oracle::hlpr::Primitives;

sub _type {
if(!defined($finance::account::trans_t::alias_tc)) {
    my $jClass = Oracle::Java::ClassLdr->loadClass (
  "finance/accountPackage/trans_tHelper");
    my $id = $jClass->id("$String");
    my $name = "finance::account::trans_t";

    my $origtc = finance::account::trans_type->_type();
    $finance::account::trans_t::alias_tc = CORBA::TypeCode->create_alias_tc($id,
  $name, $origtc);
    $finance::account::trans_t::alias_tc->jClass(
  "finance/accountPackage/trans_t");
    $finance::account::trans_t::alias_tc->perlClass(
  "finance::account::trans_t");
    $finance::account::trans_t::alias_tc->JNItype(
  "Lfinance/accountPackage/trans_type;");
}

return $finance::account::trans_t::alias_tc;
}

1;
```

finance/account/trans_type.pm

```
package finance::account::trans_type;

use Oracle::Java::VM qw(:TYPES);
use Oracle::Java::ClassLdr;
use Oracle::Java::Object;
use Oracle::hlpr::Excp;
use CORBA::TypeCode;
use Oracle::hlpr::Primitives;

*deposit = \0;
*withdrawl = \1;

sub _type {
if(!defined($finance::account::trans_type::enum_tc)) {
    my $jClass = Oracle::Java::ClassLdr->loadClass (
      "finance/accountPackage/trans_typeHelper");
    my $id = $jClass->id("$String");
    my $name = "finance::account::trans_type";

    my $members = [];
    $members->[0] = {};
    $members->[0]->{name} = "deposit";
    $members->[1] = {};
    $members->[1]->{name} = "withdrawl";
    $finance::account::trans_type::enum_tc = CORBA::TypeCode->create_enum_tc(
      $id, $name, $members);
    $finance::account::trans_type::enum_tc->jClass(
      "finance/accountPackage/trans_type");
    $finance::account::trans_type::enum_tc->perlClass(
      "finance::account::trans_type");
    $finance::account::trans_type::enum_tc->JNItype(
      "Lfinance/accountPackage/trans_type;");
}


return $finance::account::trans_type::enum_tc;
}

1;
```

finance/account/transaction.pm

```
package finance::account::transaction;

use Oracle::Java::VM qw(:TYPES);
use Oracle::Java::ClassLdr;
use Oracle::Java::Object;
use Oracle::hlpr::Excp;
use CORBA::TypeCode;
use Oracle::hlpr::Primitives;

use string;
use finance::account::trans_t;

sub _type {
if(!defined($finance::account::transaction::struct_tc)) {
    my $jClass = Oracle::Java::ClassLdr->loadClass (
      "finance/accountPackage/transactionHelper");
    my $id = $jClass->id("$String");
    my $name = "finance::account::transaction";

    my $members = [];
    $members->[0] = {};
    $members->[0]->{name} = "date";
    $members->[0]->{tc} = string->_type();
    $members->[1] = {};
    $members->[1]->{name} = "type";
    $members->[1]->{tc} = finance::account::trans_t->_type();
    $members->[2] = {};
    $members->[2]->{name} = "amount";
    $members->[2]->{tc} = Oracle::hlpr::prim_double->_type();
    $finance::account::transaction::struct_tc =
      CORBA::TypeCode->create_struct_tc($id, $name, $members);
    $finance::account::transaction::struct_tc->jClass(
      "finance/accountPackage/transaction");
    $finance::account::transaction::struct_tc->perlClass(
      "finance::account::transaction");
    $finance::account::transaction::struct_tc->JNItype(
      "Lfinance/accountPackage/transaction;");
}

return $finance::account::transaction::struct_tc;
}

1;
```

finance/account.pm

```
package finance::account;

use finance::account::trans_type;
use finance::account::trans_t;
use finance::account::transaction;

use Oracle::Java::VM qw(:TYPES);
use Oracle::Java::ClassLdr;
use Oracle::Java::Object;
use CORBA::Object;
use Oracle::hlpr::Marshall;
use Oracle::hlpr::Excp;
use CORBA::TypeCode;
use Oracle::hlpr::Primitives;


@ISA = qw ( CORBA::Object );

sub _type {
if(!defined($finance::account::intf_tc)) {
    my $jClass = Oracle::Java::ClassLdr->loadClass ("finance/accountHelper");
    my $id = $jClass->id("$String");
    my $name = "finance::account";

    $finance::account::intf_tc = CORBA::TypeCode->create_interface_tc($id,
      $name);
    $finance::account::intf_tc->jClass("finance/account");
    $finance::account::intf_tc->perlClass("finance::account");
    $finance::account::intf_tc->JNItype("Lfinance/account;");
}
```

```
return $finance::account::intf_tc;
}

*minbalance = \500.000000;
sub interestrate {
my $self = shift; my $jClass;
if (scalar(@_)) {
use Oracle::hlpr::Primitives;
my $interestrateJava = Oracle::hlpr::Marshall->marshal( $_[0],
  Oracle::hlpr::prim_double->_type() );
my $ret = $self->{javaObj}->interestrate("$double", $interestrateJava, "$void");
return Oracle::hlpr::Excp->throw($ret)
        if (Oracle::hlpr::Excp->isexcp($ret));
return ;
} else {
my $ret = $self->{javaObj}->interestrate("$double");
return Oracle::hlpr::Excp->throw($ret)
        if (Oracle::hlpr::Excp->isexcp($ret));
use Oracle::hlpr::Primitives;
$retNew = Oracle::hlpr::Marshall->unmarshal( $ret,
  Oracle::hlpr::prim_double->_type() );
return $retNew;
}
}

sub lasttranstype {
my $self = shift; my $jClass;
if (scalar(@_)) {
use finance::account::trans_t;
my $lasttranstypeJava = Oracle::hlpr::Marshall->marshal( $_[0],
  finance::account::trans_t->_type() );
my $ret = $self->{javaObj}->lasttranstype(
  "Lfinance/accountPackage/trans_type;", $lasttranstypeJava, "$void");
return Oracle::hlpr::Excp->throw($ret)
        if (Oracle::hlpr::Excp->isexcp($ret));
return ;
} else {
my $ret = $self->{javaObj}->lasttranstype(
  "Lfinance/accountPackage/trans_type;");
return Oracle::hlpr::Excp->throw($ret)
        if (Oracle::hlpr::Excp->isexcp($ret));
use finance::account::trans_t;
$retNew = Oracle::hlpr::Marshall->unmarshal( $ret,
  finance::account::trans_t->_type() );
return $retNew;
}
}

sub getbalance {
my $self = shift; my $jClass;
my $ret = $self->{javaObj}->getbalance("$double");
return Oracle::hlpr::Excp->throw($ret)
        if (Oracle::hlpr::Excp->isexcp($ret));
use Oracle::hlpr::Primitives;
$retNew = Oracle::hlpr::Marshall->unmarshal( $ret,
  Oracle::hlpr::prim_double->_type() );
return $retNew;
}

sub totaldeposits {
my $self = shift; my $jClass;
use Oracle::hlpr::Primitives;
my $frombankidJava = Oracle::hlpr::Marshall->marshal( $_[0],
  Oracle::hlpr::prim_string->_type() );
use string;
my $dateJava = Oracle::hlpr::Marshall->marshal( $_[1], string->_type() );
my $ret = $self->{javaObj}->totaldeposits("$String", $frombankidJava, "$String",
  $dateJava, "$double");
return Oracle::hlpr::Excp->throw($ret)
        if (Oracle::hlpr::Excp->isexcp($ret));
use Oracle::hlpr::Primitives;
$retNew = Oracle::hlpr::Marshall->unmarshal( $ret,
  Oracle::hlpr::prim_double->_type() );
return $retNew;
}

sub do_transaction {
my $self = shift; my $jClass;
use finance::account::transaction;
my $transJava = Oracle::hlpr::Marshall->marshal( $_[0],
  finance::account::transaction->_type() );
$jClass = Oracle::Java::ClassLdr->loadClass (
  "finance/accountPackage/transactionHolder");
$transHldr = Oracle::Java::Object->new ($jClass);
$transHldr->_set_field ("value",
  "Lfinance/accountPackage/transaction;", $transJava);
my $ret = $self->{javaObj}->do_transaction(
  "Lfinance/accountPackage/transactionHolder;", $transHldr, "$short");
return Oracle::hlpr::Excp->throw($ret)
```

```
        if (Oracle::hlpr::Excp->isexcp($ret));
$transOut = $transHldr->_get_field ("value",
  "Lfinance/accountPackage/transaction;");
use finance::account::transaction;
$_[0] = Oracle::hlpr::Marshall->unmarshal( $transOut,
  finance::account::transaction->_type() );
use Oracle::hlpr::Primitives;
$retNew = Oracle::hlpr::Marshall->unmarshal( $ret,
  Oracle::hlpr::prim_short->_type() );
return $retNew;
}

sub collateral {
my $self = shift; my $jClass;
use CORBA::Any;
my $assetJava = Oracle::hlpr::Marshall->marshal( $_[0], CORBA::Any->_type() );
my $ret = $self->{javaObj}->collateral("$CORBA::Any::JNItype", $assetJava,
  "$CORBA::Any::JNItype");
return Oracle::hlpr::Excp->throw($ret)
        if (Oracle::hlpr::Excp->isexcp($ret));
use CORBA::Any;
$retNew = Oracle::hlpr::Marshall->unmarshal( $ret, CORBA::Any->_type() );
return $retNew;
}

1;
```

finance/bank/accountseq.pm

```
package finance::bank::accountseq;

use Oracle::Java::VM qw(:TYPES);
use Oracle::Java::ClassLdr;
use Oracle::Java::Object;
use Oracle::hlpr::Excp;
use CORBA::TypeCode;
use Oracle::hlpr::Primitives;

sub _type {
if(!defined($finance::bank::accountseq::alias_tc)) {
    my $jClass = Oracle::Java::ClassLdr->loadClass (
  "finance/bankPackage/accountseqHelper");
    my $id = $jClass->id("$String");
    my $name = "finance::bank::accountseq";
    my $array_tc = CORBA::TypeCode->create_sequence_tc(0,
  finance::account->_type());
    $array_tc->JNItype("[Lfinance/account;");
    my $origtc = $array_tc;
    $finance::bank::accountseq::alias_tc = CORBA::TypeCode->create_alias_tc($id,
  $name, $origtc);
    $finance::bank::accountseq::alias_tc->jClass(
  "finance/bankPackage/accountseq");
    $finance::bank::accountseq::alias_tc-
>perlClass("finance::bank::accountseq");
    $finance::bank::accountseq::alias_tc->JNItype("[Lfinance/account;");
}

return $finance::bank::accountseq::alias_tc;
}

1;
```

finance/bank/badaccount.pm

```
package finance::bank::badaccount;

use Oracle::Java::VM qw(:TYPES);
use Oracle::Java::ClassLdr;
use Oracle::Java::Object;
use Oracle::hlpr::Excp;
use CORBA::TypeCode;
use Oracle::hlpr::Primitives;


sub _type {
if(!defined($finance::bank::badaccount::struct_tc)) {
    my $jClass = Oracle::Java::ClassLdr->loadClass (
      "finance/bankPackage/badaccountHelper");
    my $id = $jClass->id("$String");
    my $name = "finance::bank::badaccount";

    my $members = [];
    $finance::bank::badaccount::struct_tc =
      CORBA::TypeCode->create_exception_tc($id, $name, $members);
    $finance::bank::badaccount::struct_tc->jClass(
      "finance/bankPackage/badaccount");
    $finance::bank::badaccount::struct_tc->perlClass(
      "finance::bank::badaccount");
    $finance::bank::badaccount::struct_tc->JNItype(
      "Lfinance/bankPackage/badaccount;");
}

return $finance::bank::badaccount::struct_tc;
}

1;
```

finance/bank/stringseq.pm

```
package finance::bank::stringseq;

use Oracle::Java::VM qw(:TYPES);
use Oracle::Java::ClassLdr;
use Oracle::Java::Object;
use Oracle::hlpr::Excp;
use CORBA::TypeCode;
use Oracle::hlpr::Primitives;

sub _type {
if(!defined($finance::bank::stringseq::alias_tc)) {
    my $jClass = Oracle::Java::ClassLdr->loadClass (
  "finance/bankPackage/stringseqHelper");
    my $id = $jClass->id("$String");
    my $name = "finance::bank::stringseq";

    my $array_tc = CORBA::TypeCode->create_sequence_tc(2,
  Oracle::hlpr::prim_string->_type());
    $array_tc->JNItype("[$String");
    my $origtc = $array_tc;
    $finance::bank::stringseq::alias_tc = CORBA::TypeCode->create_alias_tc(
  $id, $name, $origtc);
    $finance::bank::stringseq::alias_tc->jClass(
  "finance/bankPackage/stringseq");
    $finance::bank::stringseq::alias_tc->perlClass("finance::bank::stringseq");
    $finance::bank::stringseq::alias_tc->JNItype("[$String");
}

return $finance::bank::stringseq::alias_tc;
}

1;
```

finance/bank.pm

```
package finance::bank;

use finance::bank::stringseq;
use finance::bank::accountseq;
use finance::bank::badaccount;

use Oracle::Java::VM qw(:TYPES);
use Oracle::Java::ClassLdr;
use Oracle::Java::Object;
use CORBA::Object;
use Oracle::hlpr::Marshall;
use Oracle::hlpr::Excp;
use CORBA::TypeCode;
use Oracle::hlpr::Primitives;

@ISA = qw ( CORBA::Object );

sub _type {
if(!defined($finance::bank::intf_tc)) {
    my $jClass = Oracle::Java::ClassLdr->loadClass ("finance/bankHelper");
    my $id = $jClass->id("$String");
    my $name = "finance::bank";

    $finance::bank::intf_tc = CORBA::TypeCode->create_interface_tc($id, $name);
    $finance::bank::intf_tc->jClass("finance/bank");
    $finance::bank::intf_tc->perlClass("finance::bank");
    $finance::bank::intf_tc->JNItype("Lfinance/bank;");
}

return $finance::bank::intf_tc;
}

*stuff = \3;
sub newaccount {
my $self = shift; my $jClass;
use Oracle::hlpr::Primitives;
my $nameJava = Oracle::hlpr::Marshall->marshal( $_[0],
  Oracle::hlpr::prim_string->_type() );
my $ret = $self->{javaObj}->newaccount("$String", $nameJava,
  "Lfinance/account;");
return Oracle::hlpr::Excp->throw($ret)
        if (Oracle::hlpr::Excp->isexcp($ret));
use finance::account;
```

```
my $retNew = Oracle::hlpr::Marshall->unmarshal( $ret,
  finance::account->_type() );
return $retNew;
}

sub newjointaccount {
my $self = shift; my $jClass;
use finance::bank::stringseq;
my $namesJava = Oracle::hlpr::Marshall->marshal( $_[0],
  finance::bank::stringseq->_type() );
my $ret = $self->{javaObj}->newjointaccount("[$String", $namesJava,
  "Lfinance/account;");
return Oracle::hlpr::Excp->throw($ret)
        if (Oracle::hlpr::Excp->isexcp($ret));
use finance::account;
my $retNew = Oracle::hlpr::Marshall->unmarshal( $ret,
  finance::account->_type() );
return $retNew;
}

sub newchecking {
my $self = shift; my $jClass;
use Oracle::hlpr::Primitives;
my $nameJava = Oracle::hlpr::Marshall->marshal( $_[0],
  Oracle::hlpr::prim_string->_type() );
my $ret = $self->{javaObj}->newchecking("$String", $nameJava,
  "Lfinance/checkingaccount;");
return Oracle::hlpr::Excp->throw($ret)
        if (Oracle::hlpr::Excp->isexcp($ret));
use finance::checkingaccount;
my $retNew = Oracle::hlpr::Marshall->unmarshal( $ret,
  finance::checkingaccount->_type() );
return $retNew;
}

sub getaccount {
my $self = shift; my $jClass;
use Oracle::hlpr::Primitives;
my $nameJava = Oracle::hlpr::Marshall->marshal( $_[0],
  Oracle::hlpr::prim_string->_type() );
my $ret = $self->{javaObj}->getaccount("$String", $nameJava,
  "Lfinance/account;");
return Oracle::hlpr::Excp->throw($ret)
        if (Oracle::hlpr::Excp->isexcp($ret));
use finance::account;
```

```
my $retNew = Oracle::hlpr::Marshall->unmarshal( $ret,
  finance::account->_type() );
return $retNew;
}

sub getaccounts {
my $self = shift; my $jClass;
$jClass = Oracle::Java::ClassLdr->loadClass (
  "finance/bankPackage/accountseqHolder");
my $accountsHldr = Oracle::Java::Object->new ($jClass);
my $ret = $self->{javaObj}->getaccounts(
  "Lfinance/bankPackage/accountseqHolder;", $accountsHldr, "$int");
return Oracle::hlpr::Excp->throw($ret)
        if (Oracle::hlpr::Excp->isexcp($ret));
my $accountsOut = $accountsHldr->_get_field ("value", "[Lfinance/account;");
use finance::bank::accountseq;
$_[0] = Oracle::hlpr::Marshall->unmarshal( $accountsOut,
  finance::bank::accountseq->_type() );
use Oracle::hlpr::Primitives;
my $retNew = Oracle::hlpr::Marshall->unmarshal( $ret,
  Oracle::hlpr::prim_long->_type() );
return $retNew;
}

sub getchecking {
my $self = shift; my $jClass;
use Oracle::hlpr::Primitives;
my $nameJava = Oracle::hlpr::Marshall->marshal( $_[0],
  Oracle::hlpr::prim_string->_type() );
my $ret = $self->{javaObj}->getchecking("$String", $nameJava,
  "Lfinance/checkingaccount;");
return Oracle::hlpr::Excp->throw($ret)
        if (Oracle::hlpr::Excp->isexcp($ret));
use finance::checkingaccount;
my $retNew = Oracle::hlpr::Marshall->unmarshal( $ret,
  finance::checkingaccount->_type() );
return $retNew;
}

1;
```

finance/checkingaccount.pm

```
package finance::checkingaccount;
```

```
use finance::account::trans_type;
use finance::account::trans_t;
use finance::account::transaction;

use Oracle::Java::VM qw(:TYPES);
use Oracle::Java::ClassLdr;
use Oracle::Java::Object;
use CORBA::Object;
use Oracle::hlpr::Marshall;
use Oracle::hlpr::Excp;
use CORBA::TypeCode;
use Oracle::hlpr::Primitives;

use finance::account;

@ISA = qw (  finance::account );

sub _type {
if(!defined($finance::checkingaccount::intf_tc)) {
    my $jClass = Oracle::Java::ClassLdr->loadClass (
      "finance/checkingaccountHelper");
    my $id = $jClass->id("$String");
    my $name = "finance::checkingaccount";

    $finance::checkingaccount::intf_tc = CORBA::TypeCode->create_interface_tc(
      $id, $name);
    $finance::checkingaccount::intf_tc->jClass("finance/checkingaccount");
    $finance::checkingaccount::intf_tc->perlClass("finance::checkingaccount");
    $finance::checkingaccount::intf_tc->JNItype("Lfinance/checkingaccount;");
}

return $finance::checkingaccount::intf_tc;
}

1;
```

finance.pm

```
package finance;

use finance::account;
use finance::checkingaccount;
use finance::bank;

*L = \3;
```

```
1;
```

outer/inner.pm

```
package outer::inner;


*thing = \3;
1;
```

outer.pm

```
package outer;

use outer::inner;

1;
```

Part II

Perl Cartridge

10

Perl Cartridge Overview

Perl is an interpreted language that is commonly used to write CGI scripts. Perl has powerful text processing capabilities, which makes it ideal for parsing requests from clients and generating dynamic HTML. You can download Perl from many sites on the Internet; you can find a list of pointers at **http://www.perl.com**.

The Perl cartridge contains a Perl interpreter that runs under the application server. Although you can run Perl scripts using Oracle Application Server without using the Perl cartridge (that is, they are run as CGI scripts), you can get better performance if you run the Perl scripts under the Perl cartridge. In addition, because the Perl cartridge has the Perl interpreter in it, you do not need to have the **perl** executable on your system.

Perl scripts written for the Perl cartridge are slightly different from Perl scripts written for a CGI environment because of how the cartridge runs the interpreter. If you already have Perl scripts on your system that you run in a CGI environment, you may need to modify them to make them run correctly under the cartridge.

The Perl cartridge is based on Perl version 5.004_01.

Note: Oracle Application Server does not support Perl interpreter executables where the SUID bit has been set.

Contents

How the Perl Cartridge Improves Performance

Perl scripts run faster under the Perl cartridge than in a CGI environment because:

- The cartridge maintains a persistent Perl interpreter.

  This avoids the overhead of allocating and constructing a new interpreter each time the server receives a request to run a Perl CGI script. The interpreter is loaded once in memory and it keeps running after handling each request.

- The cartridge pre-compiles Perl scripts and caches them.

  When the Perl cartridge receives a request, the Perl interpreter is ready to run the compiled script. It does not have to spend time compiling the script, unless the script has changed since the last time it was compiled. If so, the cartridge detects that the script has changed, and this causes it to compile the new version of the script and caches it.

Files in the Distribution

In addition to distributing the Perl cartridge, Oracle Application Server also distributes the Perl binaries, sources, and man pages. The binaries and man pages are installed when you install Oracle Application Server, but the Perl sources are not installed. You can access the source files, which are in compressed form, from the CD.

Table 10–1 Perl files in the distribution

Directory	Description
$ORAWEB_HOME/../cartx/common/perl	Top-level directory for the Perl cartridge
$ORAWEB_HOME/../cartx/common/perl/lib	Perl cartridge library file and runtime files
$ORAWEB_HOME/../cartx/common/perl/bin	Perl binaries
$ORAWEB_HOME/../cartx/common/perl/man	Perl man pages (UNIX only)
$ORAWEB_HOME/../cartx/common/perl/lib/Pod/html/pod	Perl help pages in HTML (Windows NT only). The top-level page is **perl.html**.
$ORAWEB_HOME/../cartx/common/perl/src	Perl sources (compressed). Note that this directory is not installed if you are installing from a CD.

Using $ORAWEB_HOME/../cartx/common/perl as Your Main Perl Installation

You can use the Perl distribution that is provided with Oracle Application Server as your main Perl installation and use it to run Perl scripts outside the context of the application server. For example, you can use it to run Perl scripts from a shell.

To use the Perl executables:

1. Add **$ORAWEB_HOME/../cartx/common/perl/bin** to your path.

   If you are using the C shell:

   ```
   % set path = ($ORAWEB_HOME/../cartx/common/perl/bin $path)
   ```

   If you are using the Bourne or Korn shell:

   ```
   $ PATH=$ORAWEB_HOME/../cartx/common/perl/bin:$PATH; export PATH
   ```

2. Set the PERL5LIB environment variable.

   If you are using the C shell:

   ```
   % setenv PERL5LIB $ORAWEB_HOME/../cartx/common/perl/lib/sun4-solaris/
   5.00401:$ORAWEB_HOME/../cartx/common/perl/lib:$ORAWEB_HOME/../cartx/common/
   perl/lib/site_perl/sun4-solaris:$ORAWEB_HOME/../cartx/common/perl/lib/
   site_perl
   ```

   If you are using the Bourne or Korn shell:

   ```
   $ PERL5LIB=$ORAWEB_HOME/../cartx/common/perl/lib/sun4-solaris/
   5.00401:$ORAWEB_HOME/../cartx/common/perl/lib:$ORAWEB_HOME/../cartx/common/
   perl/lib/site_perl/sun4-solaris:$ORAWEB_HOME/../cartx/common/perl/lib/
   site_perl; export PERL5LIB
   ```

To use the Perl man pages, add **$ORAWEB_HOME/../cartx/common/perl/man** to the MANPATH environment variable. If MANPATH is already set, do the following:

If you are using the C shell:

```
%  setenv MANPATH ${MANPATH}:$ORAWEB_HOME/../cartx/common/perl/man
```

If you are using the Bourne or Korn shell:

```
$  MANPATH=$MANPATH:$ORAWEB_HOME/../cartx/common/perl/man; export MANPATH
```

Note: Currently, the NT environment has a 512 byte limitation on the expanded length of some environment variables (CLASSPATH, JAVA_HOME, etc.). Some Oracle Application Server cartridges and JCO objects will try to expand environment variables. Therefore, make sure that your environment variables are not longer than 250-300 characters long.

Variations from Perl Standard Version

The Perl distribution that comes with the Oracle Application Server Perl cartridge is the standard Perl version 5.004_01. The interpreter is built as a shared object in UNIX, **libperlctx.so**, and a shared library in NT, **perlnt40.dll**. The Perl and LiveHTML cartridges link with the shared object or library at runtime.

The use of the Perl interpreter as a shared object or dynamic library instead of statically linking the interpreter with the Perl or LiveHTML cartridges allows the interpreter to be upgraded. This design also allows the Perl or LiveHTML cartridges to use another compatible Perl installation on the site.

Running scripts through the Perl cartridge differs from running scripts through a standard Perl interpreter in the following ways (applies to LiveHTML cartridge also):

- Standard I/O is redirected to the WRB client I/O, that is, to the client browser.
- STDERR is redirected to the WRB Logger.
- Additional CGI environment variables are returned to the Perl interpreter whenever it calls for system environment variables.
- `fork` call is not supported. Instead, the `system` call should be used. The `system` call modifies the implementation of the Perl interpreter to redirect child process output to the WRB client I/O.
- Support for error logging.
- Support for performance instrumentation.

You can get the standard version of Perl executables from **http://www.perl.org**.

11 Tutorial

This chapter provides step-by-step instructions on how to create a Perl application in Oracle Application Server. The tutorial steps you through the following operations:

Note that you must be able to log in as the “admin” user for Oracle Application Server to add applications to the server.

1. Writing the Perl Script

The sample Perl application runs a Perl script that displays the values of several CGI environment variables.

Type the following Perl script in a file and save it as **showEnv.pl**. Place the file in the **$ORAWEB_HOME/test** directory. If you do not have permission to create this directory, you can put the file in another directory, but remember your directory name when specifying the virtual path mapping.

```
print "Content-type: text/html\n\n";

print "<html>\n";
print "<head>\n";
print "<title>Some CGI environment variables</title>\n";
print "</head>\n";
print "<body bgcolor=white>\n";
print "<h1>Some CGI environment variables</h1>\n";

@varsToDisplay = (
    'HTTP_USER_AGENT',
    'REQUEST_METHOD',
    'PATH_INFO',
    'PATH_TRANSLATED');

print "<dl>\n";
foreach (@varsToDisplay) {
    print "<dt>$_\n<dd>$ENV{$_}\n";
}

print "</dl>\n";
print "</body></html>\n";
```

2. Creating a Perl Application and its Components

You need to log in as the “admin” user for the application server in order to perform this step.

1. Start up your browser and display the top-level administration page for Oracle Application Server. You should see “OAS Sites” at the top of the left frame.
2. Click the plus sign (+) to display the sites.
3. Click the plus sign (+) next to a site name to display the components on the site. You should see “HTTP Listeners”, “Oracle Application Server”, and “Applications”.
4. Click “Applications” to display the applications in the right frame. Do not click the plus sign (+) next to Applications because you will see a list of applications for the site in the left frame, instead of Applications in the right frame.
5. On the applications page in the right frame, click the green plus icon at the top of the page. The Add Application dialog opens.
6. In the Add Application dialog:
   - Application Type: select Perl.
   - Configure Mode: select Manual, which enables you to enter configuration data using dialog boxes. The other option, From File, assumes that you have already entered the configuration data for the application in a file.
   - Click Apply.

     This displays the Add Application dialog.

7. In the Add Application dialog:
   - Application Name: enter “showCGIvals”. This name is used to identify your application.
   - Display Name: enter “showCGIvals”. This name is used in the administration forms.
   - Application Version: enter “1.0”.
   - Click Apply.

     When you click Apply, you get a Success dialog box, which contains a button that enables you to add cartridges to the application.
8. In the Success dialog box, click the Add Cartridges to Application button. This displays the Add A Cartridge dialog.
9. In the Add A Cartridge dialog:
   - Cartridge Name: enter “cart1”. This name is used to identify your Perl cartridge in your “showCGIvals” application.
   - Display Name: enter “cart1”. This name is used in the administration forms.
   - Click Apply.

Specifying a Virtual Path for Your Perl Cartridge

Before you can invoke an application from a browser, you need to associate its cartridges with virtual paths. The “showCGIvals” application that you just added contains one cartridge.

1. Click the + sign next to “showCGIvals” in the tree structure. This displays Configuration and Cartridges.
2. Under Cartridges, click the + sign next to “cart1” to display Configuration. Configuration contains Virtual Path and Cartridge Parameters.
3. Click Virtual Path to display the Virtual Path page in the right frame.
4. On the Virtual Path page, add a virtual path for the “cart1” cartridge. Set the virtual path as **/perl/test** and the physical path **%ORAWEB_HOME%/test**.
5. Click Apply.

3. Reloading

After reconfiguring Oracle Application Server, you have to reload the server for the new configuration to take effect. See “Application Administration” in the *Administration Guide*.

You also have to stop and restart the listener for the new virtual path to take effect. See “Application Administration” in the *Administration Guide*.

4. Creating an HTML Page to Invoke the Perl Script

To run the **showEnv.pl** Perl script, type the following URL in your browser:

```
http://<host>:<port>/perl/test/showEnv.pl
```

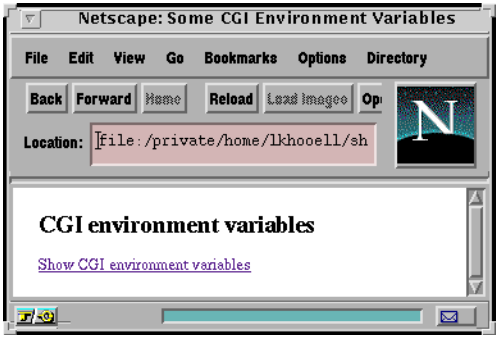
host and *port* identify the listener that knows about the cartridge. This is any listener on the application server except the node manager listener (which runs on port 8888 by default). For example, you can use the administration utility listener, which runs on port 8889 by default.

It is more common, however, to invoke the procedure from an HTML page. For example, the following HTML page has a link that calls the URL.

```
    <HTML>
    <HEAD>
    <title>CGI Environment Variables</title>
    </HEAD>
    <BODY>
    <H1>CGI Environment Variables</H1>
    <p><a href="http://hal.us.oracle.com:9999/perl/test/showEnv.pl">Show CGI
environment variables</a>
    </BODY>
    </HTML>
```

The following figures show the source page (the page containing the link that invokes the **showEnv.pl** script), and the page that is generated by the script.

Figure 11–1 The source page and the dynamically generated page in the tutorial

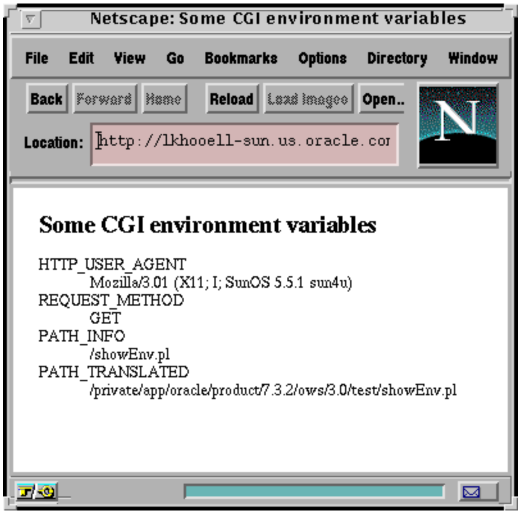


file:/private/home/lkhooell/sh

CGI environment variables

Show CGI environment variables

Figure 11–2 The page generated by the Perl script



http://lkhooell-sun.us.oracle.cor

Some CGI environment variables

```
HTTP_USER_AGENT
        Mozilla/3.01 (X11; I; SunOS 5.5.1 sun4u)
REQUEST_METHOD
        GET
PATH_INFO
        /showEnv.pl
PATH_TRANSLATED
        /private/app/oracle/product/7.3.2/ows/3.0/test/showEnv.pl
```

12

Adding and Invoking Perl Applications

To configure Perl applications, you use the Oracle Application Server Manager, which is a collection of administrative forms. On these forms you provide information such as virtual paths for the Perl applications, the minimum and maximum number of instances of Perl applications and their cartridges, and protection for the virtual paths.

Contents

Adding Perl Applications

To add Perl applications to the application server, you perform these steps:

- Add the application
- Add cartridge(s) to the application

To add an application and a cartridge:

1. Start up your browser and display the top-level administration page for Oracle Application Server.
2. Click the ⊞ next to a site name to display the components on the site. You should see “Oracle Application Server”, “HTTP Listeners”, and “Applications”.

3. Click “Applications” to display the applications in the right frame. Do not click the ⊞ next to Applications because you will see a list of applications for the site in the left frame, instead of Applications in the right frame.
4. On the applications page in the right frame, click ✚. This pops up the Add Application dialog.
5. In the Add Application dialog box:
   - Application Type: select PERL.
   - Configure Mode: select Manual, which enables you to enter configuration data using dialog boxes. The other option, From File, assumes that you have already entered the configuration data for the application in a file.
   - Click Apply.

   The Add Application dialog box opens.
6. In the Add Application dialog box:
   - Application Name: enter the name of your application as it should appear in the configuration file.
   - Display Name: enter the name that is used in the administration forms.
   - Application Version: enter the version of your application.
   - Click Apply.

   A Success dialog box opens.
7. In the Success dialog box, click the Add Cartridges to Application button. This displays the Add A Cartridge dialog box.
8. In the Add A Cartridge dialog box:
   - Cartridge Name: enter the name of your cartridge as it should appear in the configuration file.
   - Display Name: enter the name that is used in the administration forms.
   - Virtual Path: enter a path for the Perl cartridge such that users can specify this path in URLs to invoke the Perl cartridge. This path is mapped to the physical path that you specify below. See the section Virtual Paths Form below.
   - Physical Path: enter the physical directory path that leads to files for your Perl cartridge, including files for your Perl application. The virtual path specified above maps to this physical path.

9. Click Apply.
10. Stop and restart the listeners and other components of the application server.

    See “Stopping and Starting the Application Server” in the “Application Configuration” chapter for details.

> **Note:** To get your application to appear in the navigational tree, shift-click the browser’s Reload button.

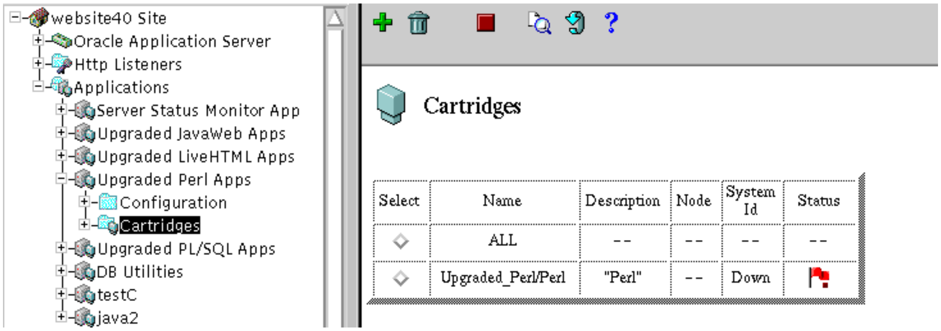
Adding Cartridges to an Existing Application

A Perl application can have one or more cartridges. You need more than one cartridge in a Perl application if you need to configure different values for the cartridge parameters. For example, you can specify a different initialization script for each cartridge.

To add a cartridge to a Perl application:

1. Select “Cartridges” under the Perl application to which you want to add cartridges in the navigational tree.

Figure 12–1 Adding Perl cartridges to an existing application



Select	Name	Description	Node	System Id	Status
◇	ALL	--	--	--	--
◇	Upgraded_Perl/Perl	"Perl"	--	Down	🚩

2. Click ✚ to display the Add Cartridge dialog.
3. In the Add Cartridge dialog:
   - Configure Mode: select Manually.
   - Click Apply, which displays the Add A Cartridge dialog.
4. In the Add A Cartridge dialog:

- Cartridge Name: enter the name that the server uses to identify your Perl cartridge in your application.
- Display Name: enter the name that is used in the administration forms.
- Virtual Path: enter a path for the Perl cartridge such that users can specify this path in URLs to invoke the Perl cartridge. This path is mapped to the physical path that you specify below. See the section Virtual Paths Form below.
- Physical Path: enter the physical directory path that leads to files for your Perl cartridge, including files for your Perl application. The virtual path specified above maps to this physical path.

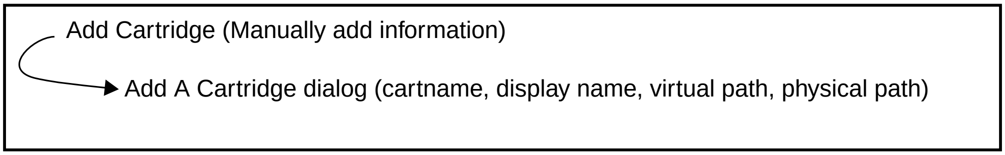
> **Note:** For security reasons, you cannot specify a physical path ending with "`..`". But you can use "`..`" in the physical path setting to indicate an upper directory level. For example, "`/routines/../libraries/`".

- Click Apply.

> **Note:** To get your new cartridge to appear in the navigational tree, shift-click the browser's Reload button.

The following figure summarizes the dialog boxes that you completed. The fields in the dialog boxes are listed in parentheses.

Figure 12–2 Dialogs to add Perl cartridges


Add Cartridge (Manually add information)
Add A Cartridge dialog (cartname, display name, virtual path, physical path)

Configuring Perl Applications

The configuration forms are divided into two sections: application configuration and cartridge configuration. Forms in the application configuration section contain parameters that apply for the entire application, while forms in the cartridge configuration secton contain parameters that apply to a particular cartridge.

Application Configuration

Application configuration parameters are described in the “Application Configuration” chapter, because they are the same for all types of applications.

Cartridge Configuration

For Perl cartridges, the cartridge configuration section contains two forms: the Virtual Paths form and the Perl Parameters form.

Virtual Paths Form

The Virtual Paths form enables you to specify a virtual path for a Perl cartridge. Users can then specify this virtual path in URLs to invoke the cartridge. The virtual path is available from all listeners listed in the Web Configuration page for the application.

For example, if you specify a virtual path of **/myApp** for a Perl cartridge, users can invoke the cartridge by typing **/myApp/***file*, where *file* is a Perl script that can be found in the physical paths associated with the **/myApp** virtual path.

Perl Parameters Form

The Perl Parameters form (Figure 12–3) enables you to define parameters specific to the Perl cartridge. The form contains the following parameters:

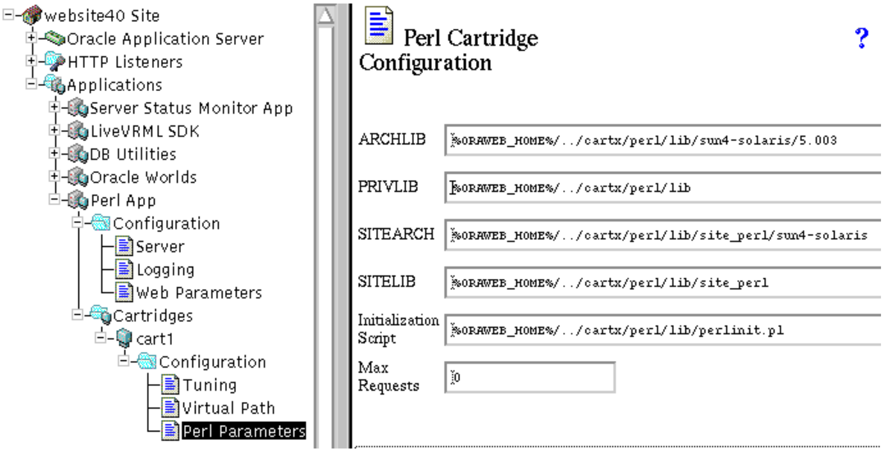
Table 12–1 Cartridge parameters

Name	Value
ARCHLIB	The path for architecture-dependent libraries. **Default value:** %ORAWEB_HOME%/../cartx/common/perl/lib/sun4-solaris/5.00401
PRIVLIB	The path for private libraries. **Default value:** %ORAWEB_HOME%/../cartx/common/perl/lib
SITEARCH	The path for site-specific architecture-dependent libraries. **Default value:** %ORAWEB_HOME%/../cartx/common/perl/lib/site_perl/sun4-solaris
SITELIB	The path for site-specific libraries. **Default value:** %ORAWEB_HOME%/../cartx/common/perl/lib/site_perl

Table 12–1 Cartridge parameters

Name	Value
Initialization Script	The script that is run when an instance of the Perl cartridge starts up. **Default value**: %ORAWEB_HOME%/../cartx/common/perl/lib/perlinit.pl
Max Requests	The number of requests that a cartridge server handles before it terminates. This field can be useful while you are developing Perl applications. If you call a Perl library, the Perl interpreter caches the Perl library and uses the cached version for subsequent requests. If you modify the library, you want the interpreter to load the new version. To do this, you have to terminate the cartridge server process so that a new cartridge server process (with a new Perl interpreter) would handle the request. A quick way of doing this is to set the Max Requests value to 1. **Default**: There is no default, which means that the cartridge server can handle an unlimited number of requests.

Figure 12–3 Perl Cartridge Parameters form



ARCHLIB	%ORAWEB_HOME%/../cartx/perl/lib/sun4-solaris/5.003
PRIVLIB	%ORAWEB_HOME%/../cartx/perl/lib
SITEARCH	%ORAWEB_HOME%/../cartx/perl/lib/site_perl/sun4-solaris
SITELIB	%ORAWEB_HOME%/../cartx/perl/lib/site_perl
Initialization Script	%ORAWEB_HOME%/../cartx/perl/lib/perlinit.pl
Max Requests	0

Number of Requests Processed by a Cartridge Instance

Some AUTOLOAD subroutines can cause the Perl cartridge's symbol table to grow as the cartridge handles each additional request. To limit the growth, set Max Requests to a small value.

Invoking Perl Cartridges

To invoke a Perl script under the Perl cartridge, the URL must be in the following format:

`http://host_and_domain_name[:port]/virtual_path/script_name[?query_string]`

where:

- *host_and_domain_name* specifies the domain and machine where the application server is running.
- *port* specifies the port at which the application server is listening. If omitted, port 80 is assumed.
- *virtual_path* specifies a virtual path mapped to a Perl cartridge.
- *script_name* specifies the file containing the Perl script. By convention, Perl scripts have a ".pl" extension. You can also specify other extensions for your Perl scripts in the Perl Parameters branch of your Perl application in the Node Manager.
- *query_string* specifies parameters for the script.

For example, if a browser sends the following URL:

http://www.acme.com:9000/perl/myScript.pl

the server running on **www.acme.com** and listening at port **9000** would handle the request. When the Listener receives the request, it passes the request to the WRB because the `/perl` virtual directory is configured to call a Perl cartridge. The Perl cartridge then executes **myScript.pl**.

Life Cycle of the Perl Cartridge

This section describes what the Perl cartridge does when it receives a request. This section assumes knowledge of the callback functions used by the application server (WRB).

You do not need to know the information in this section in order to use the Perl cartridge. However, this information is useful if you want to understand the architecture of the cartridge.

When the first instance of a Perl cartridge starts up in a cartridge server, it executes the InitRuntime callback function. The InitRuntime function constructs the Perl interpreter and runs the **$ORAWEB_HOME/../cartx/common/perl/lib/persist.pl** boot script. The function then executes the Perl script specified by the InitScript configuration parameter as a subroutine in the context of the **persist.pl** script. The default value for InitScript is **$ORAWEB_HOME/../cartx/common/perl/lib/perlinit.pl**.

The script specified by InitScript can perform initialization tasks needed by other Perl scripts. For example, you can load modules or open database connections at this point.

The Authorize callback function is executed when the cartridge needs to authorize the URL request. The Authorize function checks if the requested object is protected under any authorization schemes or restrictions. If the Authorize callback function succeeds, the Exec callback function is called next. The Exec function:

- gets the values of the CGI environment variables
- determines the Perl script to run
- determines the parameters for the procedure

The cartridge compiles the Perl script specified in the URL and stores it as a subroutine in a package whose name is based on the Perl script name. This allows the script's variables and subroutines to be localized to a package.

The requested Perl script is run as a subroutine in the context of the **persist.pl** boot script. Later, when another request for the same Perl script is received, the script is not recompiled (unless it has been modified). Instead, the package name corresponding to the Perl script name is generated and the uniquely identified subroutine in that package is called.

The Shutdown callback function is called automatically by the application server to shut down a cartridge instance. Before shutting down the instance, the Shutdown function runs the Perl script specified by the ShutScript configuration parameter. This script allows you to clean up any initialization done by the script specified by the InitScript configuration parameter. The default value for ShutScript is **$ORAWEB_HOME/../cartx/common/perl/lib/perlshut.pl**. After running the script, the Shutdown callback function deallocates and destroys the Perl interpreter.

13

Writing Perl Scripts

For Perl scripts to run correctly under the Perl cartridge, they need to follow certain rules. Note that you may need to modify existing CGI Perl scripts so that they comply with these rules.

Contents

Customized cgi-lib.pl Library

If your Perl scripts use **cgi-lib.pl** (see **http://cgi-lib.stanford.edu/cgi-lib/** for the latest information), you have to modify your scripts to use a version of the library that has been customized for the Perl cartridge. The unmodified version of **cgi-lib.pl** will not work with the Perl cartridge because the cartridge runs the Perl interpreter in a persistent manner. The modified version of **cgi-lib.pl** is **$ORAWEB_HOME/sample/perl/mycgi-lib.pl**.

Instead of writing

```
require "./cgi-lib.pl";
```

in your Perl scripts, use the following lines:

```
$ORACLE_HOME = $ENV{'ORACLE_HOME'};
$ORAWEB_HOME = "$ORACLE_HOME/ows/4.0";
require "$ORAWEB_HOME/../cartx/common/perl/sample/mycgi-lib.pl";
```

Variable Scoping

Be careful with namespace and variable scoping when running Perl scripts under the Perl cartridge. In conventional CGI scripts, you declare a variable and use it. You do not have to worry about undefining the variable because the script is restarted for each request and is not reentrant.

In the case of the Perl cartridge, global variables persist across multiple calls. The value acquired by a variable at the end of one execution of the script is the initial value for the variable when the script is executed the next time. This might cause inconsistent outputs, as seen in the following example:

```
print "Content-type: text/plain\n\n";
@question = (where, are, you, staying);
print "@question\n";
$" = "\n";
@answer = (all, on, separate, lines);
print "@answer\n";
```

When run by the cartridge for the first time, it outputs:

```
where are you staying
all
on
separate
lines
```

When run the second time and thereafter, it outputs:

```
where
are
you
staying
all
on
separate
lines
```

This is because the script changes the value of the Perl special variable $" to “\n” at line 4 and does not reset it to its original value before exiting the script. The Perl cartridge has a initial value of " " (blank) for the special variable.

One way of fixing the problem is to add this line to the end of the script to reset the value of the variable:

```
$" = " ";
```

If the execution of your script depends on values of global variables, make sure that these variables are reset to the original values.

To avoid the above problem:

- Limit the scope of your variable to the required extent only. Make sure that the variables expire after their use (by scoping the variable with `my()`) or that the script resets them to their original values.
- Reduce global variables and localize them where possible. When you need to modify Perl’s global variables, localize them so that the modification affects the local instance of the variable only. This causes the modified value of the variable to be applicable only for that run of the script.

Another way to fix the example above is to localize the $" variable:

```
1  print "Content-type: text/plain\n\n";
2  @question = (where, are, you, staying);
3  print "@question\n";
4  local($");
5  $" = "\n";
6  @answer = (all, on, separate, lines);
7  print "@answer\n";
```

Line 4 localizes the $" Perl special variable. When the script runs as a subroutine in a package, the localized variable $" has life only for that run of the script.

Namespace Collisions

The Perl cartridge caches compiled Perl scripts to speed up the response time. If not properly handled, the caching of multiple Perl scripts can lead to namespace collisions. To avoid this, the Perl cartridge translates the Perl script file name into a package name that is unique, and then compiles the code into the package using `eval`. The script is now available to the Perl cartridge in compiled form as a subroutine in the unique package name. When a request for the script is received, the cartridge translates the filename to the package name and runs the subroutine handler.

Although the above mechanism avoids the namespace collisions, you need to remember that the default package name for the script is no longer “main”. The default package name for the script is something that is generated by the cartridge.

The following example shows how the different package name affects your Perl scripts.

Perl comes with library files, which are Perl scripts that provide utility functions. For example, **bigint.pl** provides Perl with arbitrary size integer mathematics subroutines. This library defines many of the subroutines in the namespace of the package “main”. Here is an example:

```
# normalize string form of number.  Strip leading zeros.  Strip any
#   white space and add a sign, if missing.
# Strings that are not numbers result the value 'NaN'.
sub main'bnorm { #(num_str) return num_str
    local($_) = @_;
    s/\s+//g;                           # strip white space
    if (s/^([+-]?)0*(\d+)$/$1$2/) {     # test if number
        substr($_,$[,0) = '+' unless $1; # Add missing sign
        s/^-0/+0/;
        $_;
    } else {
        'NaN';
    }
}
```

A conventional CGI Perl script can use this library as follows:

```
1  # namespace.pl
2  require "bigint.pl";
3  print "Content-type: text/plain\n\n";
4  $one =  &bnorm(  456);
5  print "one = $one\n";
```

Line 4 can call the `bnorm` subroutine without specifying the package name, because the default package name for conventional scripts is “main” and the `bnorm` subroutine is available in the “main” package’s namespace. But under the Perl cartridge, a script is compiled into a package whose name depends on the filename. Any `eval` statement (note that `require` calls `eval`) is evaluated in this package’s namespace. In the example, although **bigint.pl** is compiled and stored in the package of your script, the subroutines are explicitly stored in the “main” package by fully qualified subroutine name (for example, main'bnorm).

To run the example under the Perl cartridge, you need to modify the script to call `bnorm` as `main'bnorm`.

```
# namespace.pl
require "bigint.pl";
print "Content-type: text/plain\n\n";
$one =  &main'bnorm(  456);
print "one = $one\n";
```

You should not assume that the default package name is “main”. To see the name of the package that you are currently in, you can invoke the Perl function `caller()`, which returns a list containing the package name as generated by the Perl cartridge, the file name, and the current line number.

No Need for the #! Line

Typically, the first line of Perl CGI scripts tells the system the location of the Perl interpreter. The line begins with “#!” and looks something like:

```
#!/usr/bin/perl
```

Perl scripts that are executed under the Perl cartridge do not need to have that line because the cartridge uses its own built-in Perl interpreter.

System Resources

System resources acquired by your Perl script should be freed before the script exits; otherwise, the persistent Perl interpreter in the Perl cartridge will reach system limits for the resources.

In conventional CGI Perl scripts, you can open a file and do file operations without closing it before the script exits. It does not matter in this case because the resources are returned when the Perl interpreter exits, but in the Perl cartridge environment, the file remains open even after the script execution is finished. You have to explicitly close the file in your script.

The DBI and DBD::Oracle Modules

The DBI and DBD::Oracle modules allow you to access both Oracle 7 and Oracle 8 databases. The DBI (database interface API) module provides a consistent database access API to PERL scripts independent of database type. The DBD::Oracle module allows the DBI API to access Oracle 7 and 8 databases. The DBD::Oracle module (version 0.44) that is distributed with Oracle Application Server is based on client libraries for the Oracle 8 database.

Pre-Loading Modules - Persistent Database Connections

Your Perl script may contain instructions that need not be executed repetitively for each request of the script. Performance improves if these instructions are run only once, and only the necessary portion is run for each request of the Perl script. For example, if your Perl script accesses an Oracle database using DBI and the DBD::Oracle module, you can pre-load the module and log on to the database when an instance of the Perl cartridge starts up, instead of loading the module and logging on each time the script is requested. Your script would open a cursor and fetch and return the results from the cursor. This pre-loading of a module would give you a database connection that persists across requests.

You can find the DBI and DBD::Oracle modules in the **$ORAWEB_HOME/../cartx/common/perl/lib/site_perl** directory. You can find details on DBI and DBD::Oracle at **http://www.hermetica.com/technologia/DBI/**.

To pre-load modules and perform initial tasks, edit the **$ORAWEB_HOME/../cartx/common/perl/lib/perlinit.pl** file. This file is executed only once, when the cartridge instance starts up. By default, there are no executable statements in this file. This file is specified by the InitScript parameter, the value of which can be changed in the Perl cartridge Configuration page.

The following example shows a typical Perl script that logs on to a database, makes a query, and displays the output. Assume that this script is called **scott.pl**:

```
use DBI;
print "Content-type: text/plain\n\n";
$dbh = DBI->connect("", "scott", "tiger", "Oracle") || die $DBI::errstr;
$stmt = $dbh->prepare("select * from emp order by empno") || die
                                                      $DBI::errstr;
$rc = $stmt->execute() || die $DBI::errstr;
$nfields = $stmt->rows();
print "Query will return $nfields fields\n\n";
while (($empno, $name) = $stmt->fetchrow()) { print "$empno $name\n"; }
warn $DBI::errstr if $DBI::err;
die "fetch error: " . $DBI::errstr if $DBI::err;
$stmt->finish() || die "can't close cursor";
$dbh->disconnect() || die "cant't log off Oracle";
```

Line 1 loads the DBI module. Loading the DBI module consists of reading in the Perl module script (**DBI.pm**) and the shared object (**DBI.so**). These files extend the Perl language by defining calls that enable Perl scripts to connect to Oracle databases. Once the module is loaded, the shared object is dynamically linked with the cartridge, the .pm Perl scripts are read in, compiled, and stored in the package namespace. Scripts can then make calls defined in the .pm file. The example calls the `connect()`, `prepare()`, `execute()`, `rows()`, `fetchrow()`, `finish()`, and `disconnect()` methods in the DBI module.

To change the example so that it pre-loads the DBI module and logs in to the database, add these lines to the **perlinit.pl** file:

```
1   # Contents of perlinit.pl
2   package Scott;
3   use DBI;
4   $dbh = DBI->connect("", "scott", "tiger", "Oracle") || die $DBI::errstr;;
```

Line 2 declares the package name to be Scott.

Line 3 loads the DBI module. The shared object **DBI.so** is dynamically linked with the cartridge process address space and the .pm files are compiled and stored into the Scott package’s namespace.

Line 4 calls the `connect()` method of the DBI module, which connects and logs in to the Oracle database specified by the TWO_TASK environment variable. Connection can also be made to a remote database if it is specified in the first parameter of the `connect()` call. The connection context is stored in the `$dbh` variable. For your script to access this context, you need to declare your script to be in the same package namespace (in Scott namespace) as shown below.

scott.pl contains the following:

```
# Contents of scott.pl
package Scott;
print "Content-type: text/plain\n\n";
$stmt = $dbh->prepare("select * from emp order by empno") || die
                                                               $DBI::errstr;
$rc = $stmt->execute() || die $DBI::errstr;
$nfields = $stmt->rows();
print "Query will return $nfields fields\n\n";
while (($empno, $name) = $stmt->fetchrow()) { print "$empno $name\n"; }
warn $DBI::errstr if $DBI::err;
die "fetch error: " . $DBI::errstr if $DBI::err;
$stmt->finish() || die "can't close cursor";
```

You need to ensure that the two portions of the script are compiled into the same package namespace. To do this, declare "`package` *`PackageName`*" at the beginning of both the **perlinit.pl** script and your script.

Note that the modified version of **scott.pl** does not call `disconnect()` because the database connection and login is done only at cartridge start-up. If you log off, you can no longer run the **scott.pl** script. You would have to terminate the cartridge process and start up another instance such that it executes the **perlinit.pl** script and reestablishes the database connection. The `disconnect()` is now done in the Perl script specified by the ShutScript configuration parameter. The default value is **$ORAWEB_HOME/../cartx/common/perl/lib/perlshut.pl**. To complete the example, you should modify **perlshut.pl** to contain the following lines:

```
# Contents of perlshut.pl
package Scott;
$dbh->disconnect() || die "cant't log off Oracle";
```

To get another persistent database connection for another script (for example, **kane.pl**), place the code from **kane.pl** that you want to put into **perlinit.pl** with a unique package name (for example, "Kane"). The **kane.pl** script should also declare "package Kane" before using any of the variables and subroutines initialized in **perlinit.pl**.

Testing Perl Scripts

The Perl cartridge installation provides a tool called **persistperl** that simulates Perl scripts running under the Perl cartridge. Use this tool to check that your Perl scripts will run correctly under the cartridge.

To use the **persistperl** tool:

1. Set your path variable to include the **$ORAWEB_HOME/../cartx/common/perl/bin** directory, which contains the **persistperl** binary.
2. Change the current directory to **$ORAWEB_HOME/../cartx/common/perl/lib**.
3. Invoke **persistperl** from a shell. It has the following syntax:

   `persistperl` *script_name* [*number_of_executions*]

script_name specifies the name of your Perl script to run, and *number_of_executions* specifies how many times it should be run. If omitted, *number_of_executions* defaults to 100. **persistperl** displays the output of the script on standard out, which you can redirect to a file. Check the output for any inconsistencies. In addition, the output can show if your script hit any system limits of acquiring resources after being run multiple times.

Perl Modules

This section describes the Perl modules that are provided with the Perl cartridge. Perl modules are reusable packages defined in library files. They function as classes whose methods can be used by any Perl script that imports the module. Some Perl modules are included in the standard Perl distribution (for example, DynaLoader, AutoLoader, and POSIX).

For detailed information on these packages, see the man pages for the packages, which are installed in the **$ORAWEB_HOME/../cartx/common/perl/man** directory.

For the latest versions of these packages, see **http://www.perl.com/CPAN**/. (CPAN stands for Comprehensive Perl Archive Network.)

DBI (version 0.79)

This module describes the database access API for the Perl language. DBI defines functions, variables, and conventions that provide a consistent database interface independent of the actual database being used.

Note that DBI is just an interface between the drivers and the applications. The drivers do the actual work required for database access.

The database driver (DBD::Oracle) required for accessing Oracle databases is provided with the Perl cartridge.

DBD::Oracle (version 0.44)

This module consists of the database driver for Oracle databases. Further information on the DBD::Oracle module and DBI can be found at **http://www.hermetica.com/technologia/DBI/**.

When you access databases using the DBD::Oracle module, you can get better performance by pre-loading the module and maintaining persistent database connections. See "Pre-Loading Modules - Persistent Database Connections" on page 13-6 for details.

LWP or libwww-perl (version 5.08)

This collection of Perl modules provide classes that you can use to write Web clients, parse HTML, and communicate with different types of servers, such as mail servers, HTTP servers, and NNTP servers. LWP provides three classes:

- LWP::Request

  This class is created by the client, and it specifies the request and all associated request data. If the request is to receive a document from the web, the request class is created with the URL of the document and document name.

- LWP::Response

  This class is returned by the server in response to a request. For example, if the request was for a document, the document is returned, or if the request was to send mail, the acknowledgment that the mail has been sent is returned.

- LWP::UserAgent

  This class provides the interface between the request and the response. The client creates the request and passes it to a UserAgent. The UserAgent takes care of the underlying low-level communication and handling, and passes the request to the server. It then returns the response to the client, which contains the results of the request.

CGI (version 2.36)

This module provides classes that can create HTML forms on the fly and parse their contents. It also provides an interface for parsing and interpreting query strings passed to CGI scripts. In this module, you can use the value of the previous query to initialize the form, this means that the state of the form is preserved from invocation to invocation.

MD5 (version 1.7)

This module enables you to use the RSA Data Security MD5 Message Digest algorithm from within Perl programs. This module is used by the implementation of lib-www-perl.

IO (version 1.15)

This module contains submodules to handle file, pipe, and socket I/O operations. The classes supplied with this module are:

- IO::Handle

  This is the base class for all the other IO classes. This class is not usually used directly. Other IO classes inherit from this class for their implementation.

- IO::Seekable

  This class supplies seek-based methods to the other IO classes. Methods include seek, tell and clearerr.

- IO::File

  This class inherits from IO::Handle and IO::Seekable. It provides methods specific to file handles such as open, getpos, and setpos.

- IO::Pipe

  This class provides an interface to create pipe-based communication between processes.

- IO::Socket

  This class provides the interface for socket communication. It provides methods for creating and using sockets. Methods include socket, socketpair, accept, send and recv.

Net (version 1.0502)

This module is a collection of classes that provide the user with a consistent API for the client-side implementation of various protocols. The submodules of the Net module have implemented the following protocols:

- Net::FTP - File Transfer Protocol
- Net::SMTP - Simple Mail Transfer Protocol
- Net::Time - Time and Daytime protocols
- Net::NNTP - Network News Transfer Protocol
- Net::POP3 - Post Office Protocol

To complete the installation of the Net module, you need to run a configuration script:

1. Set your path to include the **perl** executable provided with the Perl cartridge.

   If you are using the C shell:

   ```
   prompt> set path = ($ORAWEB_HOME/../cartx/common/perl/bin $path)
   ```

   If you are using the Bourne or Korn shell:

   ```
   prompt> PATH=$ORAWEB_HOME/../cartx/common/perl/bin:$PATH; export $PATH
   ```

2. Set the PERLLIB environment variable.

   If you are using the C shell:

   ```
   prompt> setenv PERLLIB $ORAWEB_HOME/../cartx/common/perl/lib:
       $ORAWEB_HOME/../cartx/common/perl/lib/platform/version:
       $ORAWEB_HOME/../cartx/common/perl/lib/site_perl:
       $ORAWEB_HOME/../cartx/common/perl/lib/site_perl/platform/
                                                         version
   ```

   If you are using the Bourne or Korn shell:

   ```
   prompt> PERLLIB=$ORAWEB_HOME/../cartx/common/perl/lib:
       $ORAWEB_HOME/../cartx/common/perl/lib/platform/version:
       $ORAWEB_HOME/../cartx/common/perl/lib/site_perl:
       $ORAWEB_HOME/../cartx/common/perl/lib/site_perl/platform/
                                                          version;
       export $PERLLIB
   ```

   Replace *platform* and *version* with the appropriate values: the *platform* directory varies between installations of Perl. For example, on Solaris, *platform* is `sun4-solaris`. The *version* directory refers to the directory named after the OS version. For example, for Solaris, it could be `5.00401`.

3. Change directory to **$ORAWEB_HOME/../cartx/common/perl/lib/site_perl/Net**, and execute the script **configure.pl**:

   ```
   prompt> cd $ORAWEB_HOME/../cartx/common/perl/lib/site_perl/Net
   prompt> perl configure.pl
   ```

   The script prompts for configuration information about the machine, and creates a Perl script called **Config.pm**, which contains the configuration information. This script is required by all the packages in the Net module.

Data-Dumper (version 2.07)

This module is required for “persistifying” Perl data structures. Given a list of scalars or reference variables, this module has methods to output them in Perl syntax. The return value can be `eval`’ed to get back the original data structure. This module is used in the implementation of some other modules supplied with the Perl cartridge, for example, net and libwww-perl.

Here is a simple example of how to use this package:

```
use Data::Dumper;
$boo = [ 1, [], "abcd", {1 => 'a', 22 => 'b'}, \\"p\q\'r"];
print Dumper($boo);          # Pretty Prints the data-structure
```

```
$bar = Dumper($boo);         # $bar has the Perl statement which when
                             # eval-ed returns the original data structure.
print Dumper(eval($bar));    # Pretty prints $boo.
```

Developing Perl Extension Modules

The Perl cartridge installation comes with the Perl interpreter runtime environment. Perl extension modules that you develop using this Perl interpreter runtime environment can be accessed by the Perl cartridge.

The Perl interpreter installation is in the **$ORAWEB_HOME/../cartx/common/perl** directory. The **$ORAWEB_HOME/../cartx/common/perl/bin** directory contains the “perl” executable. You can develop and install your extension modules under the **$ORAWEB_HOME/../cartx/common/perl/lib** directory.

For this you need to set the following environment variables:

- Set your path variable to use the **perl** executable in the **$ORAWEB_HOME/../cartx/common/perl/bin** directory.
- Set the PERL5LIB environment variable to:

  ```
  $ORAWEB_HOME/../cartx/common/perl/lib/sun4-solaris/5.00401:$ORAWEB_HOME/../
  cartx/common/perl/lib:$ORAWEB_HOME/../cartx/common/perl/lib/site_perl/sun4-
  solaris:$ORAWEB_HOME/../cartx/common/perl/lib/site_Perl
  ```

Migrating Perl Extension Modules

You might already have a Perl interpreter and have developed extension modules for that environment. To have the same extension modules be available in the Perl cartridge environment, you should rebuild the modules in the cartridge environment. For this you can use the Perl interpreter installation, which is a part of the Perl cartridge installation.

To move the Perl extension modules from your Perl interpreter installation to the Perl cartridge installation:

1. Assume that:
   - PI is your Perl Interpreter installation directory.
   - PC is your Perl cartridge installation directory, which is **$ORAWEB_HOME/../cartx/common/perl**.
   - “Mod” is your Perl module name, and a directory with the same name as the module name exists under **PI/perl/lib/site_perl/sun4-solaris/auto/**, that is, you have the directory **PI/perl/lib/site_perl/sun4-solaris/auto/Mod**.

- You have the **PI/lib/site_perl/Mod.pm** file.

2. Copy the **PI/perl/lib/site_perl/sun4-solaris/auto/Mod** directory to **PC/lib/site_perl/sun4-solaris/auto/**.
3. Copy the **PI/lib/site_perl/Mod.pm** file to the **PC/lib/site_perl** directory.
4. Update the **PC/lib/sun4-solaris/5.00401/perllocal.pod** file to reflect all the modules moved in step 2. The **perllocal.pod** file contains information on each module in your environment. Use the **PI/lib/sun4-solaris/5.00401/perllocal.pod** file as a guide on how to update the file.

Accessing the Application Server API in Perl

You can develop Perl extension modules that access the application server API. Typically, you would write wrappers to the application server APIs in the module so that you have the same junctions available to you as part of the Perl language for developing web applications in Perl. Most of the API calls need the WRBContext as the first argument. The Perl cartridge provides you access to the WRBContext of the cartridge via a Perl global variable “::WRB”. While developing the module in C, you can access the WRBContext as follows:

```
void *WRBCtx;
WRBCtx = SvIV(perl_get_sv("::WRB", FALSE));
```

14

Upgrading your Perl Interpreter

The Perl distribution that comes with the Perl and LiveHTML cartridges is the same as that available from public domain. The version that is installed with the cartridges is 5.004_01. If you wish to upgrade to a later version, follow the instructions in this chapter. The latest version of the Perl distribution can be downloaded at **http://www.perl.com**.

> **Note:** Perl interpreter versions older than 5.004_01 are not supported by Oracle.

Contents

Installing a New Interpreter

The Perl interpreter must be installed as a shared library. In UNIX for example, the interpreter is installed as `libperl.so`.

After downloading the latest version of the Perl interpreter, decompress the downloaded file into a temporary directory. Run the Configure utility with the `-Duseshrplib` option as follows:

```
prompt> Configure -Dprefix=<installation directory> -Duseshrplib
```

Follow the instructions presented to you. Continue the installation process by running the make command in the temporary directory. In UNIX:

```
prompt> make
prompt> make test
prompt> make install
```

In NT:

You can find more installation information in the README documentation that comes with the download.

For *`<installation directory>`*,you can specify a non-application server path as long as this path is reflected in your Perl or LiveHTML application's environment variables (see the next section). The interpreter that was installed with the application server can be found in `$ORACLE_HOME/ows/cartx/common/perl` for UNIX and `%ORACLE_HOME%\ows\cartx\common\perl\` for NT.

Configuring Applications to Use a New Interpreter

After the interpreter installation has completed, you need to change the configuration information of the Perl and LiveHTML applications which will use the new interpreter. Specifically, two environment variables have to be changed: `LD_LIBRARY_PATH` (`PATH` for NT) and `STD_PERLLIB`.

To change the value of `LD_LIBRARY_PATH` (on UNIX) or `PATH` (on NT):

1. In the Oracle Application Server Manager, browse to the Environment Variables form in the Configuration folder of your Perl or LiveHTML application.
2. In UNIX, locate the `LD_LIBRARY_PATH` setting and replace the phrase

   ```
   %ORACLE_HOME%/ows/cartx/common/perl/lib/sun4-solaris/5.00401/CORE
   ```

   with

   ```
   <new installation directory>/perl/lib/sun4-solaris/<new version>/CORE
   ```

3. In NT, locate the `PATH` setting and replace the phrase

   ```
   %ORACLE_HOME%\ows\cartx\common\perl\bin
   ```

   with

   ```
   <new installation directory>\bin
   ```

To change the value of STD_PERLLIB (both UNIX and NT):

1. In a shell or DOS prompt, run the following respective commands:

   UNIX:

   ```
   prompt> perl -e 'print join(":", @INC)'
   ```

   NT:

   ```
   prompt> perl -e 'print join(";", @INC)'
   ```

The STD_PERLLIB setting in the Environment Variables form will reflect the new interpreter's settings once you click the reload button.

> **Note:** Version 5.005 of the Perl interpreter requires Perl modules with XS extensions to be recompiled. Because of this and because Perl and LiveHTML cartridges have modules with XS extensions, applications written for those cartridges will not work with version 5.005 (not binary compatible).

15

Troubleshooting

Problems Invoking Your Perl Script

If your Perl script cannot be invoked:

- Make sure that the virtual path for your cartridge maps to the physical directory that contains your Perl script.
- Make sure that the application server is functioning properly. To check this, try invoking other Perl scripts and other cartridges. You can try invoking the sample Perl scripts.

Log Files

If you have enabled logging for a Perl cartridge, log messages are written to the file you specified in the Logging page (located under Configuration).

In addition, the Perl cartridge also writes any messages sent to stderr from within the Perl scripts (for example, output from `warn` and `die`) to the log file.

Unhandled Errors

The Perl cartridge runs your Perl scripts as subroutines that are `eval`'ed. If an error occurs in the script, `eval` returns the error to the Perl cartridge, which writes the error to the log file and sends an error message to the browser.

Index